HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

DIPLOMARBEIT

Herr
Simon Wartanian

Java-Technologieanalyse und Gegenüberstellung im Bereich Datenbankabstraktion und GUI-Entwicklung

2014

Fakultät **Elektro- und Informationstechnik**

DIPLOMARBEIT

Java-Technologieanalyse und Gegenüberstellung im Bereich Datenbankabstraktion und GUI-Entwicklung

Autor:
Simon Wartanian

Studiengang:
Technische Informatik (DI (FH))

Seminargruppe:
KT10wWA-F

Erstprüfer:
Prof. Dr.-Ing. Rolf Hiersemann

Zweitprüfer:
Prof. Dr. Uwe Schneider

Mittweida, März 2014

Bibliografische Angaben

Wartanian, Simon: Java-Technologieanalyse und Gegenüberstellung im Bereich Datenbankabstraktion und GUI-Entwicklung , 117 Seiten, 35 Abbildungen, Hochschule Mittweida, University of Applied Sciences, Fakultät Elektro- und Informationstechnik

Diplomarbeit, 2014

Referat

In dieser Arbeit werden die beiden – zum Zeitpunkt der Entstehung – modernen Technologien „JPA“ und „Java FX“ analysiert, getestet und anschließend mit den alten, gängigen Vorgehensweisen verglichen.

Oft wird modernen Technologien vorgeworfen, dass ihre leichte Implementierbarkeit und Übersichtlichkeit nur auf Kosten der Performance möglich ist. Daher entstand die Idee diese Arbeit zu erstellen, um im ersten Schritt die Möglichkeiten der neuen Technologien gebündelt zu ermitteln und anschließend bei der Erstellung einer Test-Applikation zu analysieren, ob die oft gehörten Vorwürfe auch der Wahrheit entsprechen.

Zum Vergleich wird die Test-Applikation ein zweites Mal ohne modernen Technologien – unter Verwendung von Java Swing (Standard-GUI-Bibliothek), Standard-Outputs (als Logging-Methode) und direktem Datenbank-Zugriff via JDBC-Treiber – erstellt.

Es werden Analysen in den Bereichen Code-Qualität, Applikations-Usability und Performance durchgeführt, um einen objektiven Einblick in die Vorgehensweisen der einzelnen Methoden zu ermöglichen.

I. Inhaltsverzeichnis

Inhaltsverzeichnis I
Abbildungsverzeichnis II
Tabellenverzeichnis III
Quellcodeverzeichnis IV
Abkürzungsverzeichnis V
Vorwort VI
1 Einleitung 1
1.1 Inhalt und Aufbau 1
1.2 Beschreibung der Test-Applikation 2
1.3 Ziele der Diplomarbeit 3
1.3.1 Muss-Ziele 3
1.3.2 Kann-Ziele 4
2 Technologieanalyse 7
2.1 Datenbank Abstraktion via JPA 7
2.1.1 Einführung in JPA 8
2.1.2 JPA-POJOs und Properties 8
2.1.3 Annotations in JPA 10
2.1.3.1 Tabellen Annotations (vgl. [MW12], S. 34f) 11
2.1.3.2 Property Annotations 11
2.1.3.3 Primärschlüsselverwaltung via Annotations 17
2.1.3.4 Assoziationen via Annotations 22
2.1.4 Datenbankanbindung in JPA 26
2.1.4.1 Konfigurationen mittels persistence.xml 26
2.1.4.2 Anwendung der Datenbankanbindung 27
2.1.5 Abfragesprache JPQL 28
2.1.5.1 Statements mittels JPQL 29
2.1.5.2 NamedQueries in JPA (vgl. [MW12], S. 181) 30
2.1.6 Lebenszyklus eines JPA-POJOs 30

2.2 Grafische Oberfläche mittels Java FX 2 33
2.2.1 Stage, Scenes und Nodes 34
2.2.2 Komponenten und Listener 35
2.2.2.1 Text Element (vgl. [JLW14], S. 213ff) 36
2.2.2.2 Textfield Element (vgl. [JLW14], S. 213ff) 37
2.2.2.3 Checkbox Element (vgl. [JLW14], S. 213ff) 38
2.2.2.4 ChoiceBox Element (vgl. [JLW14], S. 213ff) 39
2.2.2.5 Button Element (vgl. [JLW14], S. 213ff) 41
2.2.3 Tabellen in Java FX 41
2.2.3.1 Tabellenaufbau (vgl. [JLW14], S. 205ff) 42
2.2.3.2 Text-Spalten (vgl. [JLW14], S. 205ff) 43
2.2.3.3 Zahlen-Spalten 45
2.2.3.4 Datum-Spalten 45
2.2.3.5 Checkbox-Spalten 46
2.2.3.6 Auswahl-Spalten 47
2.2.4 Diagramme in Java FX 48
2.2.4.1 Diagramm-Werte (vgl. [JLW14], S. 323) 48
2.2.4.2 Diagramm-Achsen 49
2.2.4.3 Balkendiagramm erzeugen 49
2.2.5 Layouting in Java FX 50
2.2.5.1 FlowPane, VBox und HBox in Java FX (vgl. [Org14]) 51
2.2.5.2 BorderPane in Java FX (vgl. [Org14]) 52
2.2.5.3 GridPane in Java FX (vgl. [Org14]) 53
2.2.5.4 AnchorPane in Java FX (vgl. [Org14], vgl. [Gor]) 54
3 Systemkonzept 57
3.1 Vergleichsparameter 57
3.1.1 Entwicklungsvergleich 57
3.1.1.1 Vergleichsparameter bei Entwicklung 58
3.1.2 Laufzeitvergleich 59
3.1.2.1 Vergleichsparameter der Performance 59
3.1.2.2 Vergleichsparameter Release 60

3.1.2.3 Vergleichsparameter des UI Interface 60
3.1.3 Erweiterbarkeitsvergleich 61
3.2 Anwendungsspezifikation 61
3.2.1 Beschreibung der Datenbank 62
3.2.2 Anforderung Panel Transaktionsliste 64
3.2.3 Anforderung Diagramm-Panel 66
3.2.4 Anforderung Summen-Panel 66
3.3 Vorgehensweise der Entwicklung 66
4 Entwicklung 67
4.1 Vorbereitung und Service-Klassen 67
4.1.1 Logging in den Applikationen 68
4.1.1.1 DO-Entwicklung 68
4.1.1.2 DM-Entwicklung 68
4.1.2 Exceptions in den Applikationen 68
4.1.2.1 DS-Entwicklung 68
4.1.3 Eventhandling in den Applikationen 69
4.1.3.1 DS-Entwicklung 69
4.2 Entwicklung der Datenbankabstraktion 70
4.2.1 Datenbankanbindung in den Applikationen 70
4.2.1.1 DO-Entwicklung 70
4.2.1.2 DM-Entwicklung 70
4.2.2 Primary-Key Verwaltung in den Applikationen 71
4.2.2.1 DO-Entwicklung 71
4.2.2.2 DM-Entwicklung 71
4.2.3 Datenbankzugriff in den Applikationen 71
4.2.3.1 DO-Entwicklung 71
4.2.3.2 DM-Entwicklung 74
4.3 Entwicklung der Oberfläche 74
4.3.1 Tabellen Transaktionsliste 74
4.3.1.1 DO-Entwicklung 74
4.3.1.2 DM-Entwicklung 76

4.3.2 Diagramm-Panel ... 77
4.3.2.1 DO-Entwicklung ... 77
4.3.2.2 DM-Entwicklung ... 78
4.3.3 Summen-Panel ... 80
4.3.3.1 DO-Entwicklung ... 80
4.3.3.2 DM-Entwicklung ... 80
4.3.4 Zusammenführung der Komponenten ... 81
4.3.4.1 DO-Entwicklung ... 81
4.3.4.2 DM-Entwicklung ... 82
4.4 Erweiterung der Applikation ... 82
4.4.1 Aufwand bei Datenbankänderung ... 83
4.4.1.1 DO-Entwicklung ... 83
4.4.1.2 DM-Entwicklung ... 83
4.4.2 Aufwand bei alphabetischer Sortierung der Komponenten ... 83
4.4.2.1 DO-Entwiclung ... 83
4.4.2.2 DM-Entwicklung ... 84
5 Technologievergleich ... 85
5.1 Erkenntnisse Entwicklungsvergleich ... 85
5.1.1 Datenbankabstraktion ... 85
5.1.1.1 Beschreibung der Entwicklung ... 85
5.1.1.2 Quantität der Entwicklung ... 86
5.1.1.3 Qualität der Entwicklung ... 87
5.1.1.4 Analysequalität ... 87
5.1.1.5 Fazit ... 88
5.1.2 Tabelle Transaktionsliste ... 88
5.1.2.1 Beschreibung der Entwicklung ... 88
5.1.2.2 Quantität der Entwicklung ... 89
5.1.2.3 Qualität der Entwicklung ... 89
5.1.2.4 Analysequalität ... 90
5.1.2.5 Fazit ... 90
5.1.3 Diagramm-Panel ... 90

5.1.3.1 Beschreibung der Entwicklung 90
5.1.3.2 Quantität der Entwicklung 91
5.1.3.3 Qualität der Entwicklung 91
5.1.3.4 Analysequalität 92
5.1.3.5 Fazit 92
5.1.4 Summen-Panel 92
5.1.4.1 Beschreibung der Entwicklung 92
5.1.4.2 Quantität der Entwicklung 93
5.1.4.3 Qualität der Entwicklung 93
5.1.4.4 Analysequalität 94
5.1.4.5 Fazit 94
5.1.5 gesamte Applikation 94
5.1.5.1 Beschreibung der Entwicklung 94
5.1.5.2 Quantität der Entwicklung 95
5.1.5.3 Qualität der Entwicklung 96
5.1.5.4 Analysequalität 96
5.1.5.5 Fazit 97
5.2 Erkenntnisse Laufzeitvergleich 98
5.2.1 Datenbankabstraktion 98
5.2.1.1 Performance 98
5.2.1.2 Fazit 100
5.2.2 Tabelle Transaktionsliste 100
5.2.2.1 Performance 100
5.2.2.2 Oberfläche 102
5.2.2.3 Fazit 102
5.2.3 Diagramm-Panel 102
5.2.3.1 Performance 102
5.2.3.2 Oberfläche 103
5.2.3.3 Fazit 104
5.2.4 Summen-Panel 104
5.2.4.1 Performance 104

5.2.4.2 Oberfläche 105
5.2.4.3 Fazit 105
5.2.5 gesamte Applikation 106
5.2.5.1 Performance 106
5.2.5.2 Oberfläche 107
5.2.5.3 Release 107
5.2.5.4 Fazit 108
5.3 Erkenntnisse Erweiterbarkeitsvergleich 109
5.3.1 Datenbankänderung 109
5.3.1.1 Quantität der Entwicklung 109
5.3.1.2 Qualität der Entwicklung 109
5.3.1.3 Fazit 109
5.3.2 Sortierung der Konten in Java 109
5.3.2.1 Quantität der Entwicklung 109
5.3.2.2 Qualität der Entwicklung 110
5.3.2.3 Fazit 110
6 Zusammenfassung der Ergebnisse und Ausblick 111
6.1 Zusammenfassung 111
6.1.1 Erkenntnisse bezüglich JPA 111
6.1.2 Erkenntnisse bezüglich Java FX 112
6.1.3 gemeinsame Verwendung 113
6.2 Ausblick 114
Literaturverzeichnis 115

II. Abbildungsverzeichnis

1.1 Schritte der Diplomarbeit 2
2.1 Beispiel Entitäten 7
2.2 Text mit Java FX 37
2.3 Textfeld mit Java FX 38
2.4 Checkbox mit Java-FX 39
2.5 ChoiceBox mit Java-FX 40
2.6 Button mit Java-FX 41
2.7 Java FX TableView mit verschiedenen Datentypen 42
2.8 Balkendiagramm mittels Java-FX 50
2.9 FlowPane in Java FX 52
2.10 HBox in Java FX 52
2.11 VBox in Java FX 52
2.12 BorderPane in Java FX 53
2.13 AnchorPane in Java FX 55
3.1 zu erzeugende Applikation 62
3.2 Datenbankmodell 63
4.1 fertige DO-Applikation 81
4.2 fertige DM-Applikation 82
5.1 Vergleich der Entwicklungszeit bei der Datenbankabstraktion 86
5.2 Vergleich der Qualität bei der Datenbankabstraktion 87
5.3 Vergleich der Entwicklungszeit bei der Tabellenentwicklung 89
5.4 Vergleich der Qualität bei der Tabellenentwicklung 90
5.5 Vergleich der Entwicklungszeit bei der Diagrammentwicklung 91
5.6 Vergleich der Qualität bei der Diagrammentwicklung 92
5.7 Vergleich der Entwicklungszeit bei der Summen-Anzeigeentwicklung 93
5.8 Vergleich der Qualität bei der Summen-Anzeigeentwicklung 94
5.9 Vergleich der Entwicklungszeit bei der gesamten Applikation 95
5.10 Vergleich der Qualität bei der gesamten Applikation 96

5.11 Amortisation der Einarbeitungszeit in JPA und Java FX 98

5.12 Performancevergleich der Dauer bei Datenbankabstraktion 99

5.13 Performancevergleich der Dauer beim Tabellenaufbau 101

5.14 Performancevergleich der Dauer beim Diagrammaufbau 103

5.15 Performancevergleich der Dauer beim Summen-Panel-Aufbau 105

5.16 Performancevergleich der Dauer beim Applikation-Aufbau 106

5.17 Performancevergleich des Speicherverbrauchs beim Applikation-Aufbau 107

III. Tabellenverzeichnis

1.1 Technologien bei Erstellung der Test-Applikationen 1
2.1 @Basic Attribute 13
2.2 @Enumerated Attribute 15
2.3 @Column Attribute 16
2.4 Generator-Annotations für Primärschlüssel (vgl. [MW12], S. 28) 19
2.5 Table-Generator Annotations ([MW12], S. 31) 21
2.6 Beziehungs-Annotations (vgl. [MW12], S. 90) 22
2.7 JPA Query Varianten (vgl. [MW12], S. 178f) 28
2.8 JPA Query Ergebniszugriff (vgl. [MW12], S. 179f) 29
3.1 Tabelle Transaktion und deren Spalten 64
3.2 Beschreibung der Spalten der Transaktionsanzeige 65
4.1 Exceptions in Testapplikation 69
4.2 Listener in Testapplikation 69
4.3 Datenbankanbindung in DO-Applikation 70
4.4 DO-Tabellenklassen 75
5.1 Vergleich der Entwicklungszeit bei der Datenbankabstraktion 86
5.2 Vergleich der Qualität bei der Datenbankabstraktion 87
5.3 Vergleich der Entwicklungszeit über die gesamte Applikation 95
5.4 Vergleich der Qualität bei der gesamten Applikation 96
5.5 Performance-Vergleich der Dauer bei Datenbankabstraktion 99
5.6 Performance-Vergleich des Ressourcenverbrauchs bei Datenbankabstraktion 100
5.7 Performance-Vergleich der Dauer beim Tabellenaufbau 101
5.8 Performance-Vergleich des Ressourcenverbrauchs beim Tabellenaufbau 101
5.9 Performance-Vergleich der Dauer beim Diagrammaufbau 102
5.10 Performance-Vergleich des Ressourcenverbrauchs beim Diagrammaufbau 103
5.11 Performance-Vergleich der Dauer beim Summen-Panel-Aufbau 104
5.12 Performance-Vergleich des Ressourcenverbrauchs beim Summen-Panel-Aufbau 105
5.13 Performance-Vergleich der Dauer beim Applikation-Aufbau 106

5.14 Performance-Vergleich des Ressourcenverbrauchs beim Applikation-Aufbau 106

5.15 Benötigte Jar-Archive DM-Applikation 108

IV. Quellcodeverzeichnis

2.1 Beispiel POJO mit Properties 9
2.2 Tabellen Annotationen 11
2.3 nicht persistente Properties 12
2.4 java.util.Date als Text in der Datenbank 14
2.5 Property als Enumeration 15
2.6 @Column-Annotation in Beispiel-POJO 17
2.7 Primärschlüsselvergabe via UUID 18
2.8 Primärschlüsselvergabe via Table-Generator 21
2.9 1:1 Beziehung 23
2.10 Bidirektionale 1:1 Beziehung 24
2.11 @ManyToOne Property 24
2.12 @OneToMany Property 25
2.13 @ManyToMany Property 25
2.14 Persistence Konfigurationsfile (vgl. [War13a], S. 19) 27
2.15 JPA Verbindungsaufbau (vgl. [War13a], S. 20) 27
2.16 Abfragebeispiel mittels SQL 29
2.17 Abfragebeispiel mittels JPQL (vgl. [MW12], S. 188f) 29
2.18 NamedQueries am Beispiel Person 30
2.19 transient- zu managed-object 31
2.20 detached objects in JPA 32
2.21 removed objects in JPA 33
2.22 erzeuge neues Fenster mittels Java FX 34
2.23 erzeuge neue Scene mittels Java FX (vgl. [JLW14], S. 22 - 26) 35
2.24 erzeuge ein Text-Node 36
2.25 Textmanipulation mittels Property 37
2.26 erzeuge ein Textfeld-Node mit Listener 38

2.27 erzeuge eine Checkbox-Node mit Handler 39
2.28 erzeuge eine ChoiceBox-Node mit Listener 40
2.29 erzeuge eine Button-Node mit Handler 41
2.30 erzeuge eine TextView-Node 42
2.31 ein StringProperty im Tabellen-POJO 43
2.32 erzeugen einer Textspalte im TableView 44
2.33 Textfeld-Spalte im TableView editierbar gestalten 44
2.34 Zahlen-Spalte im TableView konvertieren 45
2.35 Datum-Spalte im TableView konvertieren 45
2.36 BooleanProperty im TableView POJO 46
2.37 CheckBox-Spalte im TableView konvertieren (vgl. [Mar12]) 47
2.38 ObjectProperty mit enum-Generic 47
2.39 CheckBox-Spalte im TableView konvertieren 48
2.40 Erzeuge Series für ein BarChart 49
2.41 erzeuge Series für ein BarChart 49
2.42 erzeuge BorderPane in Java FX 53
2.43 erzeuge GridPane in Java FX 54
2.44 erzeuge AnchorPane in Java FX 54
4.1 DO Loggeraufruf 68
4.2 DM Loggeraufruf 68
4.3 DO Datensuche 72
4.4 DO Datenerzeugung und Speicherung 73
4.5 DM Datenbankzugriff 74
4.6 DO Tabellenimplementierung 75
4.7 DM JPA-POJO mit Java FX-Property 76
4.8 DO Diagrammimplementierung 78
4.9 DO Query für Diagramm 78
4.10 DM Query für Summenanzeige 79
4.11 DM Query für Summenanzeige 79

4.12 DO Query für Summenanzeige 80
4.13 DM Query für Summenanzeige 80
4.14 DM intelligenter Query-Output für Summenanzeige 81
4.15 DO Sortierung der Konten 84
4.16 DM Sortierung der Konten 84

V. Abkürzungsverzeichnis

BLOB ..... Binary Large Object, Seite 14
CD ..... Compact Disc, Seite 2
CLOB ..... Character Large Object, Seite 14
DB ..... Database, Seite 11
DM ..... Entwicklung - Deplomarbeit Modern, Seite 67
DO ..... Entwicklung - Diplomarbeit Old, Seite 67
DS ..... Entwicklung - Diplomarbeit Shared, Seite 67
EJB ..... Enterprise JavaBeansJava Persistence Query Languege, Seite 111
EJB ..... Enterprise JavaBeans, Seite 7
ID ..... Identifier, Seite 11
Java EE ..... Java Platform, Enterprise Edition, Seite 26
Java SE ..... Java Platform, Standard Edition, Seite 26
JDBC ..... Java Database Connectivity, Seite 26
JPA ..... Java Persistence API, Seite 7
JRE ..... Java Runtime Environment, Seite 59
JVM ..... Java Virtual Machine, Seite 60
ORM ..... Object-Relational Mapping, Seite 8
POJO ..... Plain Old Java Object, Seite 8
SQL ..... Structured Query Language, Seite 7
UUID ..... Universally Unique Identifier, Seite 18

VI. Vorwort

Die hier entstandene Diplomarbeit ist an der Hochschule Mittweida im Rahmen eines Studiums der technischen Informatik entstanden und wurde im Sommersemester 2014 geschrieben.

Die Motivation für das gewählte Thema – Java-Technologieanalyse und Gegenüberstellung im Bereich Datenbankabstraktion und GUI-Entwicklung – war die berufliche Erfahrung mit historisch gewachsenen Java-Systemen, bei denen auf eine Migration von aktuelleren Technologien aufgrund angeblicher Performance-Defizite verzichtet wird.

Es gilt zu ermitteln, welche Unterschiede zwischen der klassischen Entwicklung ohne modernen Technologien und Entwicklung unter Berücksichtigung dieser Technologien entstehen.

In dieser Hinsicht wird das Hauptaugenmerk auf die wichtigsten Eigenschaften einer historisch wachsenden und somit umfangreichen Software gelegt.

Hier sind die wichtigste Punkte:

- Codequalität
- Erweiterbarkeit
- Performance
- Flexibilität

Getestet werden Technologien, in den für große Anwendungen notwendigen Punkte:

- Datenbankabstraktion
- grafische Oberfläche

Um das notwendige Wissen der neuen Technologien zu ermitteln wird in den Technologien „JPA“ und „Java FX“ recherchiert.

Anschließend entsteht eine Test-Anwendung die parallel unter – und ohne Berücksichtigung der modernen Technologien geschrieben wird (moderner und klassischer Weg).

Die entstandenen Produkte werden anschließend in den oben beschriebenen Punkten analysiert, getestet und bewertet.

Das Ziel der Diplomarbeit ist es eine belegte Meinung über die Vor- und Nachteile moderner Technologien im Streitfall einbringen zu können und diese auf dem Weg zu dieser Meinung näher kennenzulernen.

1 Einleitung

1.1 Inhalt und Aufbau

Die Diplomarbeit soll aufzeigen, ob sich Entwicklung mit Java unter Verwendung moderner Technologien lohnt. Im ersten Schritt wird eine Technologieanalyse durchgeführt bei der für die Umsetzung relevante Eigenschaften ermittelt und festgehalten werden. Anschließend soll mithilfe dieser Technologien eine Test-Software entstehen. Hier wird anhand des aus der Recherche erworbenen Wissens die Umsetzung der Applikation beschrieben und exemplarisch aufgezeigt. Parallel dazu wird die gesamte Software noch einmal ohne Verwendung von gängigen Technologien geschrieben. Als Basis für diese eigene Lösung dienen die Erkenntnisse aus dem Praxisprojekt (siehe [War13b]).

	klassische Entwicklung	**moderne Entwicklung**
Logging	Standard-Output	Log4J
Datenbank-Zugriff	Direktzugriff via JDBC-Treiber	JPA
GUI	Java Swing	Java FX

Tabelle 1.1: Technologien bei Erstellung der Test-Applikationen

Das Testapplikationskonzept aus Kapitel 3 enthält Komponenten damit folgende Punkte zur Analyse zur Verfügung stehen:

- Verhalten des Layouts in der Anzeige
- Verhalten einer komplexen Tabelle mit verschiedenen Datentypen
- Verhalten der Buttons
- Verhalten von Warndialogen beim unerlaubten Schließen der Applikation

Nach der Fertigstellung der Applikationen werden exemplarisch Code-Segmente für das selbe Anwendungsgebiet verglichen, um hier – trotz gleicher Programmiersprache – die starken Differenzen aufzuzeigen. Das Hauptaugenmerk beim Aufbau des Quellcodes liegt auf der Analyse der folgenden Punkte:

- Übersichtlichkeit
- Erweiterbarkeit
- Entwicklungsaufwand

Im Anschluss wird die fertige Software mit Testdaten befüllt und anschließend auf ihre Usability und Performance getestet. Hier wird sich zeigen, ob die verwendeten Tech-

nologien die gewünschte Performance erbringen, oder ob der herkömmliche Weg doch der Bessere ist.

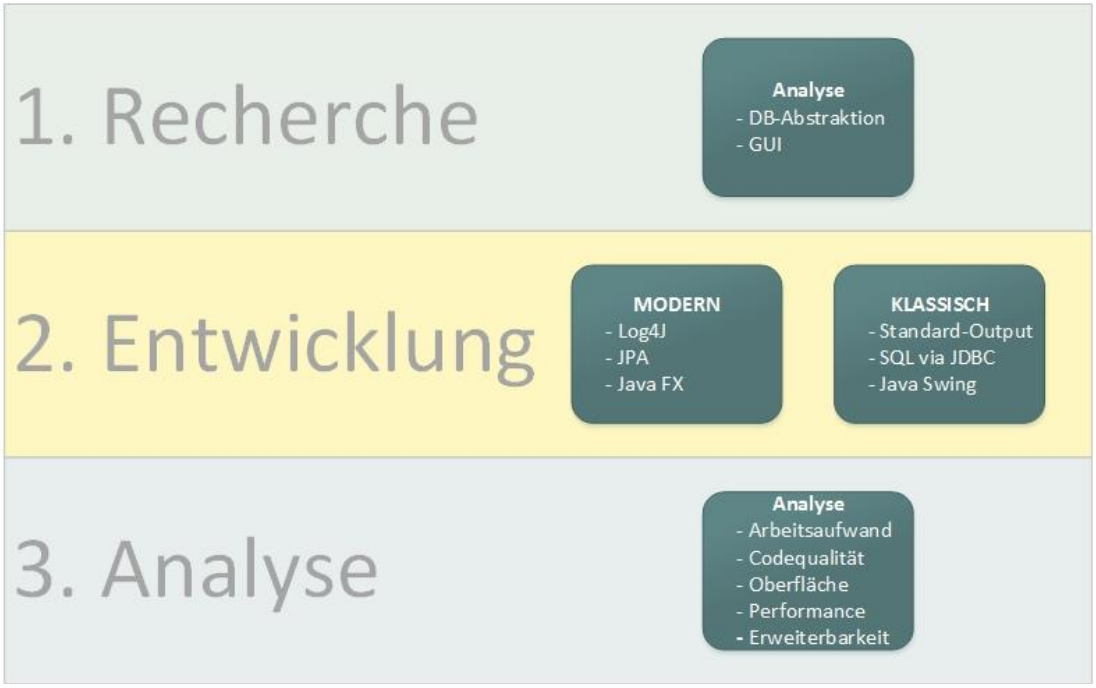


Abbildung 1.1: Schritte der Diplomarbeit

Die Anhänge in diesem Buch – als auch das dazugehörige Verzeichnis – befinden sich im Anschluss an die Diplomarbeit.

Den vollständige Sourcecode, etwaige Analyseklassen und eine leere Testdatenbank befinden sich auf einer CD auf der letzten Seite dieses Buches.

1.2 Beschreibung der Test-Applikation

Als Test-Applikation wird eine einfache Transaktions-Software entwickelt. Die Software soll aus einem Frame bestehen, in dem eine Tabelle mit Transaktionen (analog zu Listen aus Online-Banking) befüllt wird. In dieser Tabelle soll das selektive Ändern der Daten möglich sein. Zum Testen von Assoziationen soll jede Transaktion einer verursachenden Person zugeordnet werden können. Es sollen über einen Dialog neue Transaktionen hinzugefügt werden können.

Eine genauere Beschreibung der zu erstellenden Applikation befindet sich in Kapitel 3. Dort wird erst die Applikation geplant und anschließend mit der Umsetzung in Kapitel 4 begonnen.

1.3 Ziele der Diplomarbeit

Die Ziele der Diplomarbeit liegen in erster Linie darin die Technologien „JPA“ und „Java FX“ in den wichtigen Bereichen kennenzulernen und zu analysieren. Anschließend soll ermittelt werden, welche Vor- und Nachteile solche Technologien gegenüber herkömmlichen Entwicklungsmethoden haben.

1.3.1 Muss-Ziele

Als Muss-Ziele werden jene Ziele deklariert, die auf jeden Fall erfüllt werden müssen. Als Schwerpunkt der Diplomarbeit wird die Recherche gesehen, somit ist diese auch im „Muss“-Bereich deklariert.

Ziele der Untersuchung von JPA

1. Recherchieren und exemplarisches Testen der Assoziationsmethodik
2. Recherche und exemplarisches Testen des Lebenszyklus einer Entität in JPA
3. Recherche der Zugriffsarten über die Sprache „JPQL“

Ziele der Untersuchung von Java FX

1. Recherche und exemplarisches Testen der Anwendung von einzelnen Kommunikationskomponenten[1]
2. Recherche und exemplarisches Testen von Layout-Managern
3. Recherche und exemplarisches Testen von komplexen Tabellen
4. Recherche und exemplarisches Testen der Anwendung von Handlern und Listenern

Ziele des Aufbaus der Test-Applikation

1. Es muss eine externe Datenbank zum einheitlichen Testen der beiden Applikationen aufgesetzt werden.
2. Es muss ein Datenbankmodell entstehen, das einen sinnvollen Vergleich der Assoziationen anhand der Applikationen ermöglicht.
3. Es muss eine Oberfläche entstehen, die das Vergleichen in den folgenden Punkten ermöglicht:

[1] Textfelder, Auswahlfelder, Buttons, usw.

- Tabellen
- Dialoge
- Diagramme
- Buttons

Ziele der Abschluss-Analyse

Es muss eine Analyse in den folgenden Punkten erfolgen:

- Code-Analyse (Arbeitszeit und Qualität)
- Usability-Analyse
- Performance-Analyse (Rechenzeit und Ressourcenverbrauch)
- Erweiterbarkeit (Aufwand und Dauer)

Da allerdings (noch) nicht klar ist, welche Erkenntnisse bei der Analyse errungen werden, ist es schwer diese Tests näher zu definieren. Eine genauere Definition befindet sich unter den „Kann“-Zielen.

1.3.2 Kann-Ziele

In diesem Bereich werden die optionalen Ziele – großteils für den Abschluss-Test nach Fertigstellung der beiden zu vergleichenden Applikationen – beschrieben. Die hier erwähnten Punkte befinden sich nicht unter den Muss-Zielen, da es schwierig ist vorherzusagen, wie sich die technische Recherche und Umsetzung auf die anschließenden Tests auswirkt. Hier werden lediglich erste Ideen und Anreize definiert, wie der Abschlusstest ablaufen soll – falls die technische Recherche und Umsetzung wie erwartet keine Probleme aufweist.

Ziele der Performance-Analyse

1. Es soll eine Performance-Analyse anhand des Tools „JVisualVM“ entstehen.
2. Es soll eine Performance-Analyse anhand von Logging-Files entstehen.

Ziele der Code-Analyse

1. Analyse der Einarbeitungs- und Entwicklungszeit anhand einer geführten Zeitaufzeichnung (siehe Anhang 3)
2. Erweiterbarkeit – Analyse anhand eines Fallbeispiels
3. Analyse der Qualität anhand des Analysetools „SonarQube“ in den Punkten

    - Statement-Anzahl
    - Klassenanzahl
    - Zeilenanzahl
    - Komplexitätsgrad

Ziele der Usability-Analyse

1. analysiere das Verhalten der Anzeige bei verschiedenen Fenstergrößen
2. analysiere die optische Ansprechbarkeit der Oberfläche und unterschiedlicher Reaktionen
3. teste die Oberfläche auf der Suche nach möglichen Fehlern

2 Technologieanalyse

In diesem Kapitel werden die beiden Technologien JPA und Java FX systematisch analysiert. Es werden alle Punkte untersucht, die für die Entwicklung einer gängigen Rich-Client-Anwendung notwendig sind. Gegebenenfalls werden kleine Tests durchgeführt und relevante Stellen aus dem Beispielcode präsentiert. Für sämtliche Tests wird eine externe MySQL[2] Datenbank verwendet, die sich gemeinsam mit dem vollständigen Quellcode auf einer CD im Anhang befindet.

2.1 Datenbank Abstraktion via JPA

Um direkte Datenbank-Manipulationen quer über den gesamten Programmcode zu vermeiden, ist es üblich in Java eine Art Abstraktion der Datenbank über Java-Klassen zu erstellen. Im Praxisprojekt wurden der Standard EJB und eine eigene Lösung verglichen. Da im Praxisprojekt nach Technologien für ein Framework einer Offline-Rich-Client-Anwendung gesucht wurde, und EJB eine komplexe Enterprise-Technologie – mit der dazugehörigen Client-/Server-Verbindung – darstellt, ist die Entscheidung auf die eigens entwickelte Lösung die direkt via JDBC-Treiber SQL-Statements abgibt gefallen. Diese einfache, eigens-entwickelte Lösung wird mit der standardisierten Technologie JPA[3] verglichen. (vgl. [War13b], S. 14 - 22)

Es wird schrittweise über das gesamte Kapitel hinweg versucht die Tabellen aus Abbildung 2.1 in einer Datenbank mittels JPA richtig darzustellen.

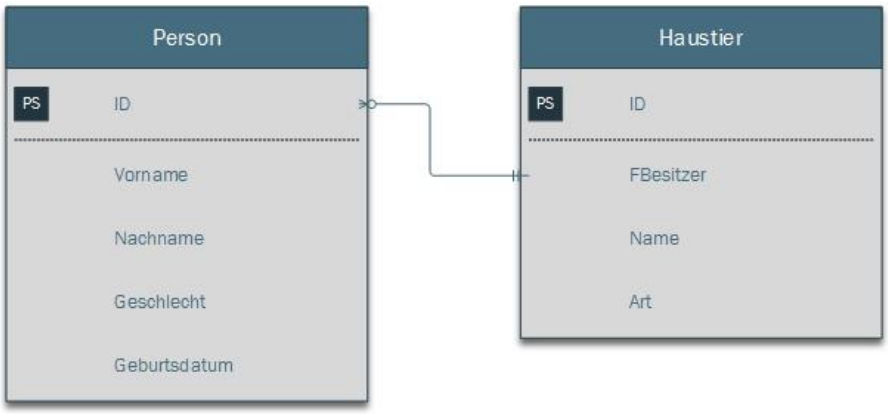


Abbildung 2.1: Beispiel Entitäten

Diese Beispiele und weitere Test-Methoden zu den folgenden Unterkapitel befinden sich auf einer CD im Anhang.

[2] siehe http://www.mysql.de/
[3] Java Persistance API

2.1.1 Einführung in JPA

JPA ist eine Java-Technologie die es ermöglicht Entitäten aus relationalen Datenbanken als POJO[4] Klassen darzustellen. Diese Abbildung nennt man OR-Mapping[5]. (vgl. [MW12], S. 13)

Im standardisierten Bereich ist JPA zum Zeitpunkt der Diplomarbeit nur wenig bekannt. Es wird in den meisten Fällen immer noch das Konkurrenzprodukt Hibernate verwendet. Allerdings stellt Hibernate eine externe Ressource dar. JPA ist seit Java-Enterprise Version 6 fester Bestandteil der Java Runtime. Abgesehen davon raten auch Experten zur zukünftigen Nutzung von JPA. (vgl. [MW12], S. 17)

Es gibt verschiedene Provider, welche das Arbeiten mit JPA ermöglichen. JPA selbst ist auf die Implementierung dieser Provider angewiesen. In diesem Projekt wird mit dem Provider Eclipse-Link gearbeitet. Eclipse-Link ist eine Technologie, welche stark mit der verwendeten Entwicklungsumgebung Eclipse verbunden ist. (vgl. [MW12], S. 17f)

2.1.2 JPA-POJOs und Properties

Je Tabelle in der Datenbank wird eine einfache Java-Klasse – die lediglich aus Properties und Annotations – besteht, erstellt. Die für diese Arbeit erforderlichen Annotations werden im Folgekapitel erarbeitet und sind daher hier noch nicht berücksichtigt.

Hierbei handelt es sich um einfache Java Klassen - mit Standardkonstruktor - die lediglich aus Properties bestimmten Datentypes bestehen und als Datenspeicher dienen. Mögliche benötigte Logik sollte sich an einem anderen Ort befinden.

Die hier erlaubten Datentypen beschränken sich auf

- Primitive Datentypen und Strings
- Serialisierbare Datentypen, einschließlich Wrapper der primitiven Datentypen, und benutzerdefinierte serialisierbare Typen
- BigInteger und BigDecimal des Package java.math
- Date und Calender des Package java.util
- Date, Time und Timestamp des Package java.util
- Enumerations
- Entity-Typen und Collections von Entity-Typen
- Eingebettete Klassen
- Collections von einfachen Typen und eingebettete Klassen

[4] Plain Old Java Object
[5] Object-Relational-Mapping

Außerdem gilt die Regel, dass jedes Property (und somit das gesamte POJO) serialisierbar sein muss. Das bedeutet, dass eine Übertragung zwischen verschiedenen Systemen gewährleistet ist.

```
...
public class Person {

  /**
   * eine private-Variable speichert den Wert intern
   */
  private String nachname;

  /**
   * eine private-Variable speichert den Wert intern
   */
  private String vorname;

  /**
   * eine private-Variable speichert den Wert intern
   */
  private String geschlecht;

  /**
   * das Geburtsdatum der Person
   */
  private Date geburtsdatum;

  /**
   * erlaubt das Manipulieren von aussen
   * @param nachname - der Nachname der gesetzt werden soll
   */
  public void setNachname(String nachname){
    this.nachname = nachname;
  }

  /**
   * gibt den gespeicherten Nachnamen zurueck
   * @return - gespeicherter Wert wird retouniert
   */
  public String getNachname(){
    return nachname;
  }
...
  /**
   * Standard-Konstruktor muss in einem POJO vorhanden sein
   */
  public Person(){
    setNachname(""); // ein moeglicher Default-Wert kann hier gesetzt werden
  }
}
```

Quellcode 2.1: Beispiel POJO mit Properties

Der Quellcode 2.1 beschreibt ein POJO welches eine Person darstellt. Eine Person wird in diesem Fall durch Vorname, Nachnamen und ein Geschlecht (die Auswahl ist auf „männlich“ und „weiblich“ beschränkt) definiert. Es wären somit alle Spalten der Beispiel-Entität „Person“ vorhanden – bis auf die Primärschlüssel-Spalte „ID“.

Außerdem fehlt noch jegliche Verbindung dieses POJOs zur Datenbank. Die Zuordnung zu den Datenbank-Entitäten und das Verwalten des Primärschlüssels wird in JPA via

Annotations gesteuert. Eine genaue Recherche – wie hier vorgegangen wird – befindet sich im nächsten Unterkapitel.

2.1.3 Annotations in JPA

Annotationen sind eine Art der Meta-Daten, die in Java seit der Version 5.0 verwendet werden. Es gibt verschiedene Arten der Annotations (vgl. [Ora13]):

- **„Information for the compiler“** – Hier können Annotations dafür verwendet werden Entwicklungsfehler auffindbar zu machen (zum Beispiel durch Verwendung der Annotations „@Override“ mit der man eine Methode markieren kann, welche die Basis-Methode in der vererbten Klasse überschreibt. Sobald – zum Beispiel mit einer neuen Version der vererbten Klasse – diese Basis-Methode nicht mehr vorhanden ist und somit nicht mehr überschrieben werden kann, erkennt der Compiler hier aufgrund dieser Annotations den Fehler. Es können anhand dieser Annotations auch Warnungen unterdrückt werden (Annotation „@SurpressWarning“). Diese Art der Annotations werden beim kompilieren nicht übersetzt.
- **„Compile-time and deployment-time processing“** – Einzelne Technologien können diese Annotations während dem Kompilieren dazu verwenden um XML-Files, Quellcode oder weiteres zu erzeugen.
- **„Runtime processing“** – Diese Art der Annotations werden im Echtbetrieb verwendet - und somit auch übersetzt. Hier gibt es Technologien – wie zum Beispiel JPA – die über Reflection[6] auf einzelne Methoden oder Variablen zugreifen und diese im Betrieb verwenden oder befüllen.

In JPA werden „Runtime processing“-Annotations dazu verwendet um im Betrieb die einzelnen Properties mit den Werten aus der Datenbank zu befüllen. Auch die Zuordnung der POJOs und Properties zu den dazugehörigen Enitäten wird über Annotations beschrieben. Die Methode der Primary-Key-Generierung und der Foreign-Key-Zuordnung wird auch über Annotations gesteuert. Durch diese Funktionalität bleiben die einzelnen Klassen sehr übersichtlicht und leserlich. Der Overhead[7] der dadurch im JPA-Code entsteht und möglicherweise die Performance drückt, wird im Kapitel 5.2.1 analysiert.

In JPA ist sehr viel bereits vorkonfiguriert. Wenn man mit der Standard-Konfiguration zufrieden ist, muss man kaum zusätzliche Annotations einbauen. Annotations dienen hauptsächlich dazu diese Standardkonfiguration anzupassen. In den folgenden Unterkapiteln werden die einzelnen wichtigen Funktionalitäten – in denen Annotations verwendet werden – erarbeitet und getestet.

[6] Ermöglicht das Scannen von (neuen) Klassen nach übergebenen Methodennamen um diese im Echtbetrieb auszuführen ohne das der Compiler diese beim Übersetzen gefunden oder berücksichtigt hat

[7] bezeichnet in der Software-Entwicklung überflüssige Daten oder Quellcodezeilen die bei automatisierter Erzeugung oft entstehen

2.1.3.1 Tabellen Annotations (vgl. [MW12], S. 34f)

Um ein POJO als Persistent-POJO zu markieren genügen eigentlich bereit die Annotations „@Entity“ und „@Id“. Falls man allerdings möchte, dass in der eigens für JPA entwickelten Abfragesprache JPQL[8] die Entität unter einem anderen Namen gefunden wird, kann man bei der Annotation „@Entity“ das Attribute „name“ übergeben. Standardmäßig wird hier der Klassenname verwendet, was sinnvoll und übersichtlich ist. Falls aber in verschiedenen Packages POJOs mit dem selben Namen existieren, kann ein solches Attribut äußerst nützlich sein.

Während die Annotation „@Entity“ die Java-Kommunikation eines POJOs konfiguriert wird die Annotation „@Table“ dazu verwendet um die Abbildung einer Datenbanktabelle über das POJO zu konfigurieren. Standardmäßig wird auch hier davon ausgegangen, dass die Entität in der Datenbank den selben Namen trägt wie die POJO-Klasse. Falls man hier aber einen anderen Klassennamen vergeben möchte, als die Entitätsbezeichnung in der Datenbank, muss lediglich der originale Entitätsname bei der Annotation „@Table“ unter dem Attribut „name“ abgelegt werden. Auch Unique-Constraints die über mehrere Spalten einer Tabelle gehen müssen unter dieser Annotation konfiguriert werden.

```
...
@Entity(name="Person")  // Tabelle wird in JPQL unter "Person" gefunden
@Table(name="Person",   // Tabelle stellt Entitaet "Person" der DB dar
  uniqueConstraints=  // Kombination aus VORNAME und NACHNAME muss eindeutig sein
  @UniqueConstraint(columnNames={"vorname", "nachname"}))
public class Person {
...
}
```

Quellcode 2.2: Tabellen Annotationen

Der Quellcode 2.2 zeigt anhand des POJO „Person“ wie diese beiden Tabellen-Annotations verwendet werden. In diesem Unique-Constraint-Beispiel darf eine Kombination aus Vorname und Nachname nur einmal vorkommen.

2.1.3.2 Property Annotations

Eine als „private“ gekennzeichnete Klassen-Variable inklusive der dazugehörigen Getter- und Setter-Methoden – wird als Property bezeichnet. Auch hier können Annotations verwendet werden um klarzustellen, wofür ein Property steht. Meistens bildet ein Property einfach eine Spalte der Datenbank in Java ab.

[8] Java Persistence Query Language

@Access Annotation (vgl. [MW12], S. 36f)

Es gibt auch die Möglichkeit Tabellen-Spalten über „Fields“ – also Klassenvariablen die als „public“ oder „protected“ gekennzeichnet sind und auf die somit ohne Getter- und Setter-Methoden zugegriffen wird – darzustellen. Hierfür gibt es die Annotation „@Access“, deren einziges Attribut „value“ beschreibt, ob hier via „Field“ oder „Property“ auf die Daten zugegriffen wird. Im Falle dieser Arbeit wird immer mit „Properties“ gearbeitet, da es sich hier um den Standard-Zugriffstyp handelt und der Zugriff über Public-Member-Variablen kompliziert zum Debuggen ist und allgemein als schlechter (unleserlicher) Programmierstil gilt.

@Transient Annotation (vgl. [MW12], S. 37f)

Die erste Art der Properties die man über Annotations markieren kann, sind die Properties die keine Tabellenspalte abbilden – also Werte die als keine persistente Einheit betrachtet werden. Hierbei kann es sich um Variablen handeln, die für mögliche Berechnungen oder softwareinterne Läufe nötig sind, aber keineswegs in der Datenbank abgespeichert werden sollen. Solche Properties müssen mit der Annotation „@Transient“ gekennzeichnet werden.

```
...
  /**
   * Property speichert rein Java-intern wie oft (seit Start der Applikation)
   * auf den Nachnamen des POJOs zugegriffen wurde
   */
  @Transient
  private Integer zugriffNachname;

  /**
   * speichert Vorname und Nachanme in Property und gibt diesen Wert zurueck
   * @return gesamter Name als zusammenhaengender String
   */
  public String getGesamterName(){
    return gesamterName = getVorname() + " " + getNachname();
  }

  /**
   * gibt den gespeicherten Nachnamen zurueck
   * @return - gespeicherter Wert wird retouniert
   */
  public String getNachname(){
    Integer alterWert = getZugriffNachname();
    if(alterWert == null)
      alterWert = new Integer(0);
    setZugriffNachname(new Integer(alterWert.intValue() + 1));
    return nachname;
  }

  /**
   * gibt zurueck wie oft seit Start der Applikation auf diesen Nachnamen
   * zugegriffen wurde
   * @return - Wert der Zugriffsvariable
   */
  public Integer getZugriffNachname() {
    return zugriffNachname;
```

```
36    }
38    /**
      * setzt einen neuen Wert der Zugriffsvariable
40    * muss von aussen nicht erreichbar sein, daher private
      * @param zugriffNachname - der neue Wert der Variable
42    */
      private void setZugriffNachname(Integer zugriffNachname) {
44      this.zugriffNachname = zugriffNachname;
      }
46  ...
```

Quellcode 2.3: nicht persistente Properties

Der Quellcode 2.3 zeigt wie so ein Transient-Property im Fall von der Test-Entität „Person“ verwendet werden kann. Bei jedem Zugriff auf den Nachnamen wird eine Zählvariable erhöht. Dieser Wert wird nicht in der Datenbank gesichert und geht bei jedem Terminieren der Java-Applikation verloren.

@Basic Annotation (vgl. [MW12], S. 38f)

Falls ein Property ohne Annotation beschrieben ist, geht der JPA-Provider davon aus, dass dieses Property eine gleichnamige Tabellenspalte repräsentiert. Zusätzlich wird bei einem solchen nicht näher beschriebenen Property von einem Wert ausgegangen, der auch „null“-Werte enthalten darf. Falls keine „null“-Werte erlaubt sein sollen, muss das Property via Annotations konfiguriert werden. Zusätzlich wird bei Properties ohne Annotations davon ausgegangen, dass diese bereits beim Laden des POJOs aus der Datenbank ausgelesen werden. Es gibt hier die Möglichkeit die Spalte erst auszulesen, falls explizit auf diesen Wert zugegriffen wird. Es kann sein, dass zum Beispiel bei einer Person nur der Nachname benötigt wird. Ein voreiliges Auslesen des Vornamens würde nur unnötige Zeit in Anspruch nehmen. Bei einzelnen Zugriffen spielt dieser Punkt kaum eine Rolle. Allerdings beim Iterieren über mehrere Millionen Datensätze kann eine solche Konfiguration sinnvoll sein. Diese Konfiguration erfolgt optional über die Annotation „@Basic“.

Annotation „@Basic“		
Attribut	**Typ**	**Beschreibung**
fetch	FetchType	Ladeart EAGER, LAZY
optional	boolean	null erlaubt

Tabelle 2.1: @Basic Attribute

@Temporal Annotation (vgl. [MW12], S. 39)

Diese Annotation ermöglicht es ein Datumsfeld, welches in Java durch ein „java.util.Date“ Objekt repräsentiert wird, auf der Datenbank als einen Text abzulegen. Oft ist es nicht notwendig das Datum bis auf die Millisekunde genau in der Datenbank zu speichern. In solchen Fällen ist es übersichtlicher und effizienter wenn stattdessen des Datum als formatierter String gespeichert wird. Bei der Beispieltabelle Person wird das Geburtsdatum auf diese Art abgebildet.

```
...
  /**
   * das Geburtsdatum der Person
   */
  @Temporal(TemporalType.DATE)
  private Date geburtsdatum;
...
```

Quellcode 2.4: java.util.Date als Text in der Datenbank

Bei Speichern des Datums im Textformat „yyyy-dd-MM“ geht auch die äußerst wichtige Fähigkeit der Sortierung des SQL-Outputs nach dem Datum nicht verloren.

@Lob Annotation (vgl. [MW12], S. 37 - 42)

Auch zu erwähnen ist die Zuweisung der Datentypen. Während Flags, Zahlen, Zeichenketten und Datumsangaben über „java.lang.Boolean“, „java.lang.Integer“, „java.lang.String“ und „java.sql.Date“ dargestellt werden können, gibt es auch noch die Möglichkeit binäre Datenformate in der Datenbank abzulegen. Um diese richtig zu schreiben und zu lesen gibt es die Annotation „@Lob“. In dieser Arbeit wird nur die Funktionalität der langen Texte benötigt. In MySQL gibt es den Datentyp „text“. Um diesen richtig in einen Java-String zu füllen, muss dieses Property mit der Annotation „@Lob“ markiert sein. Sonstige „CLOB“[9] und „BLOB“[10] Eigenschaften werden in dieser Arbeit nicht benötigt.

@Enumerated Annotation (vgl. [MW12], S. 41f)

Für statische Domänenwerte – wie zum Beispiel Geschlecht: männlich/weiblich – unterstützt JPA Enumerations. Hier können in Java klare Einschränkungen erstellt werden, ohne für jede Trivialität eine eigene Domänen-Tabelle erstellen zu müssen. Diese Enumerations können in der Datenbank als Text oder Zahl gespeichert werden. Der Text entspricht genau dem Titel der Enumeration. Falls man die Zahlen-Variante wählt wird jeder Zahl aufsteigend eine Enumeration zugeordnet. In dieser Arbeit wird die String-Variante verwendet, da die Datenbank auch ohne die dazugehörige Applikation lesbar

[9] Character Large Object
[10] Binary Large Object

bleiben soll. Falls jeder Enumeration eine Zahl zuordnet ist, gibt es bei reinem Datenbankzugriff ohne dem dazugehörigen Java-Quellcode keine Chance herauszufinden wofür diese Zahlen stehen. Im Fall einer Domänen-Tabelle könnte man dort nachsehen. In einem solchen Fall – wo die Domäne via Enumeration im Java-Code dargestellt ist – hat man hier keine Chance, außer der Text wird selbsterklärend (also Spalteninhalt „männlich“/„weiblich“ anstatt „1“/„2“) in der Tabelle abgespeichert. Aus diesem Grund wird in dieser Arbeit auch nicht näher auf „BLOB“ Spalten – in denen man Java-Objekte serialisiert abspeichern könnte – eingegangen, da diese bei direktem Datenbankzugriff keinen lesbaren Inhalt liefern und so gegebenenfalls das Troubleshooting um ein vielfaches erschwert wird. Um eine solche Enumeration zu konfigurieren muss die Annotation „@Enumerated“ verwendet werden.

Annotation „@Enumerated“		
Attribut	**Typ**	**Beschreibung**
value	EnumType	Aufzählungsart in Datenbank ORIGINAL, String

Tabelle 2.2: @Enumerated Attribute

Falls man in der Beispiel-Entität „Person“ auf diese Art die Spalte „Geschlecht“ befüllen wollen würde, ergäbe sich der Quellcode 2.5.

```
1  ...
     /**
3     * Diese Enumeration schraenkt die Eingabemoeglichkeiten
      * bei der Spalte "geschlecht" auf diese beiden
5     * Werte ein
      */
7    public enum GESCHLECHT{
       MAENNLICH, WEIBLICH
9    }

11   /**
      * abstrahiert die Spalte "geschlecht" aus der Datenbank
13    * Enum-Wert wird als String in der Datenbank abgelegt
      */
15   @Enumerated(EnumType.STRING)
     private GESCHLECHT geschlecht;
17
     /**
19    * @return gibt das gesetzte Geschlecht zurueck
      */
21   public GESCHLECHT getGeschlecht(){
       return geschlecht;
23   }

25   /**
      * setzt den Wert Geschlecht
27    * @param geschlecht - Werte aus der Enumeration GESCHLECHT
      */
29   public void setGeschlecht(GESCHLECHT geschlecht){
       this.geschlecht = geschlecht;
31   }
...
```

Quellcode 2.5: Property als Enumeration

@Column Annotation (vgl. [MW12], S. 42ff)

Generell sollte jedes Property – welches eine Spalte einer Tabelle abbildet – die Annotation „@Column“ verwenden. Defaultmäßig wird davon ausgegangen, dass jedes Property eine gleichnamige Spalte in der Datenbank abbildet – ohne jegliche Zusatzkonfiguration. Da die Spalten in den meisten Fällen aber bestimmte Besonderheiten aufweisen, sollte allein der Übersichtlichkeit halber jedes Attribut via „@Column“ beschrieben werden.

Annotation „@Column“			
Attribut	**Typ**	**Default**	**Beschreibung**
name	java.lang.String	Property-Name	Spaltenname
unique	boolean	false	ob auf der Spalte ein unique-constraint liegt
nullable	boolean	false	ob „Null“-Werte zugelassen werden
insertable	boolean	true	ob in diese Spalte über die Applikation Werte eingetragen werden dürfen
updatable	boolean	true	ob bereits vorhandener Werte über Applikation überschrieben werden darf
column-Definition	java.lang.String	Leer-String	DLL-Anteil der Spaltendefinition
table	java.lang.String	Tabelle laut Klassenname oder @Table	ermöglicht Speichern über mehrere Tabellen über nur ein POJO
length	int	255	maximale Länge (falls Zeichenkette)
precision	int	0	Anzahl der Ziffern (falls java.math.BigDecimal)
scale	int	0	Anzahl der Nachkommastellen (falls java.math.BigDecimal)

Tabelle 2.3: @Column Attribute

Die meisten oben gelisteten Werte sind selbsterklärend. Lediglich das Attribut „column-Definition“ bedarf einer Erklärung. JPA bietet die Möglichkeit, aus den definierten POJOs heraus auf einer verbundenen Datenbank die nötigen Tabellen zu erstellen – unter Berücksichtigung der Annotations die konfiguriert wurden. Falls man nicht möchte, dass JPA selbst den create-Befehl zusammenbaut, kann man hier die Definition einer Spalte – exklusive Namen der eindeutig im Attribut „name“ enthalten ist – unter „Column-Definition“ manuell übergeben (Zum Beispiel „varchar(200) not null“). Eine sinnvolle „Column-Definition“ ist die Verwendung der Datenbank-Systemzeit, anstatt der Applikations-Systemzeit durch den Text „DATE DEFAULT CURRENT_DATE“.

Im Fall von „Person“ würden die Klassen-Variablen wie in Quellcode 2.6 beschrieben aussehen.

```
...
  /**
   * eine private-Variable speichert den Wert intern
   */
  @Column(nullable=false, length=100)
  private String nachname;

  /**
   * eine private-Variable speichert den Wert intern
   */
  @Column(nullable=false, length=50)
  private String vorname;

  /**
   * Property speichert rein Java-intern wie oft (seit Start der Applikation)
   * auf den Nachnamen des POJOs zugegriffen wurde
   * (einziges Property ohne @Column, da nicht persistent)
   */
  @Transient
  private Integer zugriffNachname;

  /**
   * das Geburtsdatum der Person
   */
  @Temporal(TemporalType.DATE)
  @Column(nullable=false)
  private Date geburtsdatum;

  /**
   * abstrahiert die Spalte "geschlecht" aus der Datenbank
   * Enum-Wert wird als String in der Datenbank abgelegt
   */
  @Enumerated(EnumType.STRING)
  @Column(nullable=false)
  private GESCHLECHT geschlecht;
...
```

Quellcode 2.6: @Column-Annotation in Beispiel-POJO

Abgesehen von den hier erarbeiteten Annotations gibt es noch folgende zwei Annotations die betrachtet werden müssen:

- Primärschlüssel-Annotations
- Assoziations-Annotations

Diese beiden Annotations benötigen eine genauere Analyse und werden daher in den folgenden beiden Unterkapiteln beschrieben.

Bis auf diese beiden Punkte ist das Beispiel JPA-POJO „Person“ bereits fertig.

2.1.3.3 Primärschlüsselverwaltung via Annotations

Es gibt verschiedenste Herangehensweisen wenn es darum geht, welcher Wert für einen Primärschlüssel verwendet wird und wer dafür verantwortlich ist, diese Spalte

bei einem neuen Datensatz zu befüllen. In dieser Diplomarbeit wird der Ansatz der **Surrogatschlüssel** angewendet.

> *„Wir raten von der Verwendung fachlicher oder sprechender Schlüssel ab und empfehlen dem Leser sogenannte Surrogatschlüssel, also Schlüssel ohne Geschäftssemantik. Dies ist mittlerweile Konsens in der modernen Software-Entwicklung[Amb03, Kar10].“,*
>
> *Harald Wehr, Bernd Müller [MW12], S. 27*

Der Schlüssel wird somit vom Anwender der fertigen Software ferngehalten und beinhaltet keinerlei Geschäftslogik. Daher können alle Datensätze die zur Geschäftslogik gehören nach Lust und Laune manipuliert werden, ohne mögliche Verpflichtungen (Constraints) zu verletzen. Auch in unserer Beispiel-Entität „Person“ wird ein Surrogatschlüssel verwendet.

Die Variable in der POJO-Klasse, welche für die Primärschlüssel-Spalte in der Datenbank steht, wird mit der Annotation „@Id“ beschrieben. Das Befüllen dieser Surrogatschlüssel soll automatisch geschehen. Hierfür gibt es verschiedene Möglichkeiten.

Schlüssel via UUID

Beim Anlegen einer Entität befüllt Java mittels eines UUID[11] einen garantiert einzigartigen generiertem Schüssel für diese Spalte.

> *„Dies erfolgt sinnvollerweise in den Konstruktoren z.B. über einen Universally Unique Identifier (UUID) oder über einen von der Fachabteilung vorgegebenen Alogrithmus.“,*
>
> *Harald Wehr, Bernd Müller [MW12], S. 26*

Diese Variante ist die einzige die ohne zusätzliche Annotations auskommt. Hier müsste beim Erzeugen der Entität im Konstruktor zum Beispiel einfach der Wert gesetzt werden (siehe Quellcode 2.7).

```
...
  /**
   * Der Primaerschluessel
   */
  @Id
  private final String ID;
```

[11] Universally Unique Identifier

```
8   /**
     * Konstruktor generiert und setzt eine UUID
10   */
   public PersonUUID(){
12   ID = UUID.randomUUID().toString();
   }
14 ...
```

Quellcode 2.7: Primärschlüsselvergabe via UUID

Diese Variante hat den Vorteil, dass Java sofort – ohne bei der Datenbank nachfragen zu müssen – den Primärschlüssel der neuen Entität kennt. Ein Nachteil ist allerdings, dass beim Erzeugen von Datensätzen außerhalb von Java kein Schlüssel mehr automatisch vergeben wird (zum Beispiel manuelle Verbindung mit der Datenbank über ein Terminal). Außerdem beinhaltet eine UUID als Primärschlüssel keinerlei Information über eine Reihenfolge der Datensätze in einer Tabelle.

Zum Beispiel könnte man bei einem Primärschlüssel, der fortlaufend bei jedem neuen Datensatz um eins erhöht wird daraus auf eine Anlage-Reihenfolge schließen. Bei UUIDs geht diese Reihenfolgen-Information zur Gänze verloren. Für diese Diplomarbeit wurde auf UUID's als Primärschlüssel verzichtet.

Schlüssel via Primärschlüsseltabelle

Falls die Datenbank Methoden zur Generierung des Primärschlüssels zur Verfügung stellt, gibt es hierfür drei verschiedene Varianten. In JPA gibt es vier verschiedene Generatoren zur Auswahl (wobei einer der Generator „AUTO“ ist, der automatisch die gewählte Methode erkennt). Diese vier Generatorarten sind in der Enumeration GenerationType definiert. Über die Annotation @Generated-Value unter dem Attribute „strategy“ kann eine dieser Methoden ausgewählt werden.

Annotation	Beschreibung
@AUTO	Überlässt der JPA-Implemetierung die Wahl des Verfahrens, in der Regel IDENTITY oder SEQUENCE. Die Wahl ist sowohl provider- als auch datenbankabhängig.
@IDENTITY	Verwendet die Datenbankfunktionalität einer selbst inkrementierenden Spalte. Dies sind z.B. die Typ-Zusätze IDENTITY bei H2, SERIAL bei Postgres und AUTO_INCREMENT bei MySQL.
@TABLE	Verwendet eine zusätzliche Tabelle zur Verwaltung des zuletzt generierten Primärschlüssels.
@SEQUENCE	Verwendet die Datenbankfunktionalität einer Sequence. Sequences werden von H2, ORACLE, Postgres und DB2 unterstützt.

Tabelle 2.4: Generator-Annotations für Primärschlüssel (vgl. [MW12], S. 28)

Für diese Diplomarbeit wird die Variante „TABLE“ verwendet, da diese Variante als Einzige von allen Datenbanken unterstützt wird. Somit könnte man ohne Umschreiben der Applikation einfach die Datenbank wechseln.

Datenbanktabelle (vgl. [MW12], S. 29 - 32)

In dieser Variante ist der Primärschlüsselwert eine Zahl, die bei jedem neuen Datensatz erhöht wird. Als Speicher für den aktuellen Stand der Zahl wird eine eigene Tabelle erstellt. Diese Speicher-Tabelle kann manuell erstellt werden und soll aus nur zwei Spalten bestehen:

1. Name des Generators (Zum Bespiel „ID für Tabelle Person“)
2. Nächster Primärschlüsselwert (wird bei jedem Auslesen durch JPA erhöht)

Der aus dieser Tabelle ausgelesene Wert wird dann anschließend beim Anlegen eines neuen Datensatzes als Primärschlüssel verwendet.

Zusätzlich dazu gibt es die Möglichkeit die Performace zu steigern, in dem man sich einen gewissen Primärschlüsselbereich reserviert. Zum Beispiel wenn man 50 Datensätze auf einmal anlegen muss, wäre es eine Performace-Verschwendung, bei jedem Datensatz erneut bei der Primärschlüssel-Tabelle neue Werte anzufordern. Um dem zu entgehen kann man festlegen, dass bei jedem Zugriff auf die Primärschlüsseltabelle, der Wert zum Beispiel um 10 erhöht werden soll. Somit wüsste man, dass man jetzt ohne neu anzufragen die nächsten 10 Datensätze anlegen kann, und jeweils selbstständig den erhaltenen Schlüssel um eins inkrementiert. Nachdem die reservierten 10 wieder verbraucht sind, müsste man wieder einen neuen 10-er Block anfragen. Somit würde sich die Anfrage-Last von 50 Anfragen auf 5 reduzieren.

Das dadurch möglicherweise Lücken entstehen (zum Beispiel wenn man nur 5 Datensätze anlegt, aber bei jeder Anfrage 10 Schlüssel unwiderruflich reserviert), ist egal. Da sowieso nur mit Surrogatschlüsseln gearbeitet wird, spielt eine lückenlos fortlaufende Reihe keine Rolle. Die Reihenfolge der Anlage bleibt auch mit Lücken eindeutig dokumentiert. Die Tabelle 2.5 beschreibt Annotations die wichtig sind um einen Table-Generator zu erstellen.

Attribut	Typ	Default	Beschreibung
name	String	–	eindeutiger Name des Table-Generators
allocation-Size	int	50	Der Wert, um den inkrementiert wird, wenn Werte alloziert werden
catalog	String	Default-Catalog	der Katalog der Tabelle

initialValue	int	1	der Anfangswert der Primärschlüsselgenerierung
pkColumnName	String	providerspezifisch	Name der Spalte, mit der die Ids für das jeweilige Entity unterschieden werden
pkColumnValue	String	providerspezifisch	Wert, der einen Primärschlüssel von einem anderen Wert unterscheidet. Wird verwendet, wenn für verschiedene Entities eine Tabelle zum Speichern des Primärschlüssels genutzt wird.
schema	String	Defaultshema	das Schema der Tabelle
table	String	providerspezifisch	Tabellenname
uniqueConstraints	UniqueConstraint[]	{}	Spaltenname(n) für Unique-Constraints
valueColumnName	String	Providerspezifisch	Name der Tabellenspalte, die den letzten generierten Wert je Entity aufnimmt

Tabelle 2.5: Table-Generator Annotations ([MW12], S. 31)

Daraus würde sich in der Beispiel-Entität „Person“ die in Quellcode 2.8 beschriebene Konfiguration ergeben.

```
...
  /**
   * Der PrimaryKey der Entitaet
   */
  @TableGenerator(
    // Name des Generators - wird fuer Zugriff via Java benoetigt
    name="myGenerator",
    // Tabelle in der DB welche die ID's handelt
    table="ID_TABLE",
    // Spalte die den Namen der ID beschreibt (Key/Value-Prinzip)
    pkColumnName="PK_SPALTE",
    // Name der ID (in Spalte PK_SPALTE, ist der Key)
    pkColumnValue="PERSON_ID",
    // Spalte in der sich die aktuelle ID befindet (Value zum Key)
    valueColumnName="ID_SPALTE",
    // bei jeder Abfrage wird die ID um 10 inkrementiert
    allocationSize=10)
  // verwende oben konfigurierten Generator
  @GeneratedValue(strategy=GenerationType.TABLE, generator="myGenerator")
  @Id
  private BigInteger id;
...
```

Quellcode 2.8: Primärschlüsselvergabe via Table-Generator

Somit ist das Beispiel-POJO „Person“ komplett fertig und könnte in dieser Form als fertige JPA-Entität eingesetzt werden.

Im nächsten Schritt werden Assoziationen erarbeitet. Das geschieht anhand der zweiten Beispiel-Tabelle „Haustier“, die in einer 1:CM-Verbindung[12] zur Person-Tabelle steht.

2.1.3.4 Assoziationen via Annotations

In JPA orientiert sich die Implementierung der Assoziationen an dem Modell der relationalen Datenbanken und nicht an der objektorientierten Entwicklung. Während in der objektorientierten Entwicklung zwischen Assoziationen, Aggregationen und Kompositionen unterschieden wird, gibt es hier nur den Begriff „Beziehung“. Diese Beziehung kann unidirektional oder bidirektional sein. (vgl. [MW12], S. 89f)

In der Datenbankmodellierung haben sich die Platzhalter 1[13], m/n[14] und c[15] durchgesetzt (wobei c-Verbindungen nicht in der Datenbank sondern durch die Software realisiert werden). Auch in JPA können c-Verbindungen nicht standardmäßig implementiert werden, sondern müssen durch den Entwickler eingebaut werden. Darum wird bei Assoziationen im JPA das c nicht berücksichtigt. Je nach Beziehung wird in JPA eine der in Tabelle 2.6 beschriebenen Annotations verwendet.

Beziehung	Annotation
1:1	@OneToOne
1:m	@OneToMany
m:n	@ManyToMany

Tabelle 2.6: Beziehungs-Annotations (vgl. [MW12], S. 90)

1:1 Beziehungen (vgl. [MW12], S. 90 - 94)

Eine 1:1 Beziehung kommt dann zustande, wenn in dem oben beschriebenen Beispiel eine Person nur genau ein Haustier haben darf. Eine solche Beziehung ist unidirektional. Man kann nur von einer der beiden Tabellen auf die andere navigieren. Sinnvoll wäre es hier zwar einen Foreign-Key in der Personen-Tabelle einzubauen, aber damit man anschließend genau dasselbe Beispiel auch für 1:m Beziehungen verwenden kann, befindet sich der Foreign-Key in der Haustier-Tabelle.

Standardmäßig geht JPA davon aus, dass der Name solcher Spalten sich aus dem Na-

[12] Beschreibt eine Verbindung, in der eine Person zwischen null und unendlich Haustieren besitzen kann.
[13] Besagt, dass es nur genau einen Datensatz auf der betroffenen Seite der Assoziation gibt – Verwendung: 1:1, 1:m, 1:cm
[14] Besagt, dass es auf dieser Seite der Assoziation zwischen null und unendlich Datensätze gibt (n wird verwendet sobald m bereits vorkommt) – Verwendung: 1:m, 1:cm, m:n, mc:n, mc:nc
[15] Ist ein Zusatz zu m oder n Assoziationen - besagt dass es mindestens einen Datensatz geben muss (somit zwischen ein und unendlich) – Verwendung: 1:cm, mc:n, mc:nc

men der zu erreichenden Tabelle plus „_“ und dem Namen deren Primärschlüsselspalte ergibt – somit in diesem Fall „person_id“. Damit man solchen Spalten aber sinnvollere Namen geben kann, lässt sich dieser mittels der Annotation **@JoinColumn** übergeben. Falls man bei der Erstellung eines Haustieres einen Besitzer setzen möchte, muss man hier einfach mittels Setter-Methode des Foreign-Key-Properties diesen übergeben. Falls man ausgehend von einem Haustier auf dessen Besitzer zugreifen möchte geht das direkt über die Getter-Methode des Properties. Die Assoziation wird automatisch von JPA aufgelöst.

Außerdem bietet JPA die Möglichkeit beim Anlegen mehrerer Objekte die in Beziehungen zu einander stehen automatisch beim Speichern einer Entität in die Datenbank alle anderen betroffenen Entitäten zu persistieren. In diesem Beispiel würde somit bei der Neuanlage einer Person und eines Haustieres das Speichern des Haustieres reichen. Dazu muss im Haustier-POJO bei der @OneToOne Annotation der Parameter **cascade** gesetzt sein – dazu siehe Quellcode 2.9. Daraus folgt, dass in dem POJO in dem der Foreign-Key enthalten ist, ein neues Property erzeugt werden muss, welches den dazugehörigen Besitzer beschreibt.

```
...
@Entity(name="Haustier")
@Table(name="Haustier")
public class Haustier {
...
  /**
   * beschreibt den Foreign-Key zum Besitzer
   *  Paramter "optional" besagt, dass jedes Haustier einen Besitzer haben muss
   *  Paramter "nullable" ist gleichbedeutend mit "optional", wird aber von
   *   EclipseLink benoetigt
   *  Parameter "cascade" erlaubt das Mitpersistieren der Entitaet Person, falls
   *   das Haustier gespeichert wird - als Typ wurde ALL gewaehlt, da hier nicht
   *   nur beim Speichern, sondern auch beim Loeschen (im Gegensatz zu PERSIST)
   *   die referenzierte Entitaet beruecksichtigt wird
   */
  @OneToOne(optional=false, cascade=CascadeType.ALL)
  @JoinColumn(name="fbesitzer", nullable=false)
  private Person fbesitzer;

  /**
   * retouniert dem gesetzten Besitzer
   * @return gibt den Wert von FBesitzer zurueck
   */
  public Person getFbesitzer() {
    return fbesitzer;
  }

  /**
   * setzen von FBesitzer
   * @param fbesitzer - die Variable die gesetzt wird
   */
  public void setFbesitzer(Person fbesitzer) {
    this.fbesitzer = fbesitzer;
  }
...
```

Quellcode 2.9: 1:1 Beziehung

Bidirektionalisieren von 1:1 Beziehungen (vgl. [MW12], S. 95)

In den objektorientierten Datenbanken gibt es den Nachteil, dass eine 1:1 Beziehung immer unidirektional ist. Somit kann man nur von einer der beiden betroffenen Tabellen die andere erreichen. JPA bietet allerdings die Möglichkeit auch 1:1 Beziehungen bidirektional zu gestalten, indem in beiden betroffenen POJOs ein Property erstellt wird.

Das Property aus dem unidirektionalen Beispiel muss dazu nicht abgeändert werden. Es muss lediglich in dem POJO Besitzer dem betroffenen Property übergeben werden, sodass die unidirektionale Verbindung hier vom Haustier ausgeht. Den Rest regelt JPA.

```
...
public class Person {
...
  /**
   * stellt bidirektionale Verbindung zum Haustier her
   * mappedBy wird der Name der betroffenen Spalte
   * im Haustier-POJO uebergeben
   */
  @OneToOne(mappedBy="fbesitzer")
  private Haustier fhaustier;
...
```

Quellcode 2.10: Bidirektionale 1:1 Beziehung

1:m Beziehungen (vgl. [MW12], S. 99)

1:m Beziehungen werden in der Regel immer als bidirektionale Verbindungen mittels JPA erstellt. Auch hier stellt diese Eigenschaft einen Vorteil gegenüber dem Modell einer relationalen Datenbank dar.

Bei dem POJO, welches die „1“-Seite der Beziehung darstellt – und somit auch den Foreign-Key enthält – muss das Entity **@ManyToOne** hinzugefügt werden.

```
...
public class Haustier {
...
  /**
   * beschreibt den Besitzer des Haustiers
   */
  @ManyToOne
  @JoinColumn(name="fbesitzer", nullable=false)
  private Person fbesitzer;
...
```

Quellcode 2.11: @ManyToOne Property

Im Gegensatz dazu muss auf der Gegenseite die Annotation **@OneToMany** eingebaut werden. Hier gibt es wieder den äußerst sinnvollen Parameter **cascade**.

```
...
public class Person {
...
  /**
   * stellt bidirektionale Verbindung zum Haustier her
   * mappedBy wird der Name der betroffenen Spalte
   * im Haustier-POJO uebergeben
   * Beim loeschen der Person werden auch alle Haustiere
   * entfernt
   */
  @@OneToMany(mappedBy="fbesitzer", cascade=CascadeType.ALL)
  private Set<Haustier> haustiere;
...
```

Quellcode 2.12: @OneToMany Property

m:n Beziehungen (vgl. [MW12], S. 106)

Solche Beziehungen werden in der Datenbank immer über eine Zwischentabelle gelöst. Es gibt eine Zwischentabelle, die zu den beiden betroffenen Tabellen jeweils eine 1:m Beziehung hält. Dadurch entsteht zwischen den beiden betroffenen Tabellen eine m:n Beziehung.

JPA bietet die Möglichkeit diese Zwischentabelle für den Entwickler gänzlich zu verbergen. Ähnlich wie bei der @OneToMany-Seite einer 1:m-Beziehung befindet sich hierbei auf beiden Seiten im POJO ein Set mit einer Menge an Assoziationen. Diese Verbindungen werden über die Annotation **@ManyToMany** gesteuert. In Quellcode 2.13 sieht man wie sich dieses Beispiel mit einer Zwischentabelle namens „BesitzerToHaustier“ verhalten würde.

```
...
public class Person {
...
  /**
   * Haustiere die zu der Person gehoeren,
   * wenn ein Haustier auch mehrere Besitzer haben kann
   */
  @ManyToMany(cascade=CascadeType.PERSIST) // Cascade nur beim Anlegen
  @JoinTable(
    // Name der Zwischentabelle
    name="BesitzerToHaustier",
    // Name der Spalte in der Zwischentabelle die Richtung Tabelle PERSON geht
    joinColumns={@JoinColumn(name="besitzer")},
    // Name der Spalte in der Zwischentabelle die Richtung Tabelle HAUSTIER geht
    inverseJoinColumns={@JoinColumn(name="haustier")}
  )
  private Set<Haustier> haustiere;
...
```

Quellcode 2.13: @ManyToMany Property

2.1.4 Datenbankanbindung in JPA

Im vorherigen Kapitel wurde erarbeitet, wie eine Datenbanktabelle auf ein POJO abgebildet werden kann. Um diese POJOs nützen zu können, muss erst eine Datenbankverbindung aufgebaut werden.

In JPA gibt es hierzu die Klasse **EntityManager**, welche die gesamte Datenbankverbindung regelt und einen konsistenten Zustand innerhalb eines Persistenzkontextes garantiert. Die Verbindung des EntityManagers wird durch die **EntityManagerFactory** eingelesen und verwaltet. Der EntityManager wird immer durch dieses Interface erzeugt. Wobei hier zwischen Java-SE und Java-EE Anwendungen unterschieden werden muss. Bei Java-EE Anwendungen wird der EntityManager vom Application-Server übergeben, bei Java-SE hingegen wie beschrieben selbst erzeugt. Die SE-Manager sind anwendungsverwaltet, die EE-Manager hingegen container-verwaltet. In dieser Arbeit werden Methoden für Standalone Rich-Client-Anwendungen beschrieben, daher wird die Java-SE Variante analysiert. Die Konfiguration der Verbindung zur Datenbank und deren Eigenschaften liest die EntityManagerFactory aus der Konfigurationsdatei **persistence.xml** aus. (vgl. [MW12], S. 55)

In diesem Kapitel wird erarbeitet, welche Schritte notwendig sind um die POJOS erfolgreich in eine existierende Datenbank zu persistieren.

2.1.4.1 Konfigurationen mittels persistence.xml

Die eigentliche Datenbankanbindung wird im File **persistence.xml** konfiguriert.

Erst wird ein „Persistence-Unit“-Element definiert, welches anschließend die EntityManagerFactory einliest. Hier müssen alle Entity-POJOS als „class“-Elemente gelistet werden die bei dieser Konfiguration berücksichtigt werden sollen. Anschließend wird die Datenbankverbindung im Element „properties“ inklusive Username und Passwort konfiguriert. Es ist auch möglich diese Konfiguration im Betrieb via Factory-Methoden-Aufrufe zu setzen. Gerade im Fall Username und Passwort könnte man das mittels einem Login-Dialog regeln. In dieser Standalone-Testapplikation wird die Datenbankverbindung statisch vollständig im Konfigurationsfile eingetragen.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/
      xml/ns/persistence/persistence_2_0.xsd">
  <!-- Erstelel Unit - wird ueber Namen angesprochen -->
  <persistence-unit name="test_jpa_unit">
    <!-- POJOS die beruecksichgt werden sollen -->
    <class>jpa.test.Person</class>
    <class>jpa.test.Haustier</class>
    <properties>
      <!-- JDBC-Treiber werden verwendet -->
      <property name="eclipselink.jdbc.batch-writing" value="JDBC"/>
      <!-- direkte SQL Statements sind zusaetzlich zu JPQL erlaubt -->
      <property name="eclipselink.jdbc.native-sql" value="true"/>
      <!-- Statements werden zwischengespeichert und nicht neu geladen -->
      <property name="eclipselink.jdbc.cache-statements" value="true"/>
      <!-- URL zur Datenbank -->
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost
          :3306/dipl_jpa_analyse"/>
      <!-- Username und Passwort -->
      <property name="javax.persistence.jdbc.user" value="diplomarbeit"/>
      <property name="javax.persistence.jdbc.password" value="diplomarbeit"/>
      <!-- Beschreibt welcher DB-Treiber verwendet wird (hier MySQL) -->
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"
          />
    </properties>
  </persistence-unit>
</persistence>
```

Quellcode 2.14: Persistence Konfigurationsfile (vgl. [War13a], S. 19)

2.1.4.2 Anwendung der Datenbankanbindung

Durch den EntityManagerFactory wird anschließend der EntityManager erzeugt. Hier können beliebig viele Transaktionen geöffnet werden, um Daten in der Datenbank zu erstellen, zu aktualisieren oder zu löschen. Ein solcher Lebenszyklus einer Entität wird in einem eigenen Kapitel 2.1.6 analysiert. Für reines Auslesen aus der Datenbank mittels Queries wird keine Transaktion benötigt. Hierfür liefert der EntityManager die notwendigen Methoden die im nächsten Unterkapitel erläutert werden.

```
...
public class TestJPA {
...
  /**
   * liest Konfiguration aus
   */
  private EntityManagerFactory emf = Persistence
    .createEntityManagerFactory("test_jpa_unit");

  /**
   * EntityManager verwaltet Datenbankverbindung - uebernimmt Konfiguration
   */
  private EntityManager em = emf.createEntityManager();

  /**
   * da hier Standalone gearbeitet wird, benoetigt es nur eine Transaktion
   */
```

```
  private EntityTransaction et = em.getTransaction();
19
  /**
21  * Konstruktor - hier werden die Tests durchgefuehrt
   */
23 private TestJPA(){
    et.begin();
25  // Datenbank-Manipulation
    et.commit();
27 }
...
```

Quellcode 2.15: JPA Verbindungsaufbau (vgl. [War13a], S. 20)

2.1.5 Abfragesprache JPQL

Es gibt mehrere Methoden mittels JPA Queries durchzuführen. Einerseits kann man in der relationalen Datenbank direkt Statements mittels nativem SQL durchführen, oder man führt die Queries auf objektorientierter Ebene mittels JPQL (Java Persistence Query Language) durch. Hierfür gibt es im bereits bekannten EntityManager die in Tabelle 2.7 beschriebenen Methoden.

Methode	Beschreibung
createNamedQuery()	Erzeugt eine JPQL-Query die unter einem Namen gespeichert wird und immer wieder abgerufen werden kann.
createNativeQuery()	Erzeugt eine SQL-Query. Das Ergebnis kann von JPA nicht automatisch in die POJOs geladen werden. Es wird daher bei der Query eine Klasse als Parameter übergeben, welche beschreibt wohin JPA das Ergebnis füllen soll. Falls beim SQL-Statement die falschen Spalten angegeben werden, kommt es zur Exception.
createQuery()	Erzeugt eine JPQL Query aus einem String. Falls es Eingabeparameter gibt, sollte die Methode setParameter() verwendet werden. Hiermit können SQL-Injections[16] vermieden werden, da JPA darauf achtet, dass die Parameter den entsprechenden Datentyp haben.

Tabelle 2.7: JPA Query Varianten (vgl. [MW12], S. 178f)

Abgesehen von diesen Eigenheiten erfolgt der Zugriff auf das Ergebnis bei allen drei Varianten gleich. Es gibt zwei Methoden, über die man auf das Ergebnis zugreifen kann (siehe Tabelle 2.8).

[16] Beschreibt eine Art des Hackens, bei der versucht wird das SQL-Statement durch einen Eingabeparameter so zu manipulieren, dass es sich selbst aushebelt und die gewünschten Daten zurückliefert.

Methode	Beschreibung
getResultList()	gibt eine Ergebnisliste mit POJOs zurück
getSingleResult()	Gibt genau ein POJO zurück. Falls allerdings mehrere Ergebnisse gefunden wurden, wird nicht das Erste zurückgegeben, sondern es kommt zu einer Exception.

Tabelle 2.8: JPA Query Ergebniszugriff (vgl. [MW12], S. 179f)

Um das Ergebnis nicht casten[17] zu müssen, sollte man die Query als **TypedQuery** durchführen. Hier wird der Ergebnis-Typ als Generic[18] angeführt.

2.1.5.1 Statements mittels JPQL

Da JPQL objektorientiert arbeitet, unterscheidet sich die Query von normalen nativen SQL-Statements. Anstatt in der Datenbank die Tabellen zu joinen, bewegt man sich über die Java-POJOs hinweg. Ein Join funktioniert hier also nicht über den Schlüsselvergleich, sondern man betritt eine Assoziation direkt über ein Property. Falls man zum Beispiel alle Haustiere für eine Person „Max Mustermann“ haben wollen würde, ergebe sich folgender Unterschied:

Mittels SQL müsste man wegen der unidirektionalen Beziehung auf beide Tabellen zugreifen, sowohl Haustier als auch Person.

```
select h.*
from person p, haustier h
where h.fbesitzer = p.id
and p.nachname = 'Mustermann'
and p.vorname = 'Max'
```

Quellcode 2.16: Abfragebeispiel mittels SQL

Wenn die Abfrage mittels JPQL durchgeführt wird, kann man auf Objektebene die Bidirektualität nützen und kann direkt auf das entsprechende Property zugreifen.

```
select p.haustiere
from Person p
where p.nachname = 'Mustermann'
and p.vorname = 'Max'
```

Quellcode 2.17: Abfragebeispiel mittels JPQL (vgl. [MW12], S. 188f)

Wobei hier noch zu erwähnen ist das JPQL im Gegensatz zu SQL case-sensitiv arbeitet.

[17] bezeichnet das Umwandeln eines Datentypes in einen anderen – funktioniert nur wen das Objekt dazu geeignet ist

[18] kann bei neuen Java-Framework-Klassen übergeben werden, sodass bei etwaigen Rückgabewerten das Ergebnis bereits den richtigen Typ besitzt und nicht erst gecastet werden muss

2.1.5.2 NamedQueries in JPA (vgl. [MW12], S. 181)

Die letzte noch erwähnenswerte Erkenntnis sind die NamedQueries. Hier ist es möglich eine Query – die oft im Quellcode verwendet wird – unter einem Namen zur Wiederverwendung abzuspeichern. Die Anlage dieser Queries erfolgt im dazu passenden POJO. Die Anlage ist auch über die ORM.XML-Datei möglich. NamedQueries werden über die gleichnamige Annotation **@NamedQueries** konfiguriert. Diese Queries werden wie in Quellcode 2.18 gezeigt beschrieben.

```
...
@NamedQueries({
  // sucht wie im oben beschriebenen Beispiel nach Haustieren zur Person,
  // Name wird uebergeben
  @NamedQuery(
    name="searchHaustiere",
    query=  "select p.haustiere from Person p " +
        "where p.nachname=:nachname and p.vorname=:vorname"),
  // sucht eine Person - Parameter ist der Name
  @NamedQuery(
    name="searchPersonByName",
    query=  "select p from Person p " +
        "where p.nachname=:nachname and p.vorname=:vorname")
})
@Entity(name="Person")  // Tabelle wird in JPQL unter "Person" gefunden
@Table(name="Person",   // Tabelle stellt Entitaet "Person" der DB dar
  uniqueConstraints=  // Kombination aus VORNAME und NACHNAME muss eindeutig sein
  @UniqueConstraint(columnNames={"vorname", "nachname"}))
public class Person {
...
```

Quellcode 2.18: NamedQueries am Beispiel Person

2.1.6 Lebenszyklus eines JPA-POJOs

Wie im vorangegangenen Kapitel 2.1.4 beschrieben, wird die gesamte Verbindung zur Datenbank über den EntityManager gesteuert. Das Auslesen der Daten hat keinen Einfluss auf den Lebenszyklus einer Entität. Wohingegen man beim Erstellen, Ändern und Löschen von Daten immer darauf achten muss, dass bei der gesamten Transaktion keine Fehler auftreten. Darum gibt es hier die Klasse **EntityTransaction** mit welcher man eine Transaktion beginnen (begin), verüben (commit) aber auch zurückrollen (rollback) kann.

Abgesehen von dem Zustand einer Transaktion befindet sich eine Entität unabhängig davon immer in einem der folgenden vier Zustände:

- **Transient objects** sind Objekte die bereits erzeugt wurden (zum Beispiel new Person()), aber noch nicht an den EntityManager übergeben wurden.
- **Managed objects** wurden bereits an den EntityManager übergeben und sind zum Zeitpunkt der Betrachtung auch in der Datenbank persistent. Diese Objekte kön-

nen entweder synchron oder dirty[19] sein.

- **Detached objects** sind zum Zeitpunkt der Betrachtung zwar von einem EntityManager losgelöst, haben aber in der Datenbank ein Äquivalent – welches möglicherweise bereits von einem anderen EntityManager verändert wurde. Man könnte auch sagen diese Objekte laufen Gefahr nicht mit der Datenbank-Entität synchron zu sein.
- **Removed objects** sind Objekte die zwar noch mit dem EntityManager verbunden sind und auch in der Datenbank noch vorhanden sind, allerdings bereits zum Löschen vorgemerkt wurden.

Überführen von transient- zu managed-objects (vgl. [MW12], S. 57ff)

Um ein transient-object zum Zustand „managed“ zu überführen, muss man es an den EntityManager übergeben. Damit ist es zwar zum Speichern vorgemerkt, wird aber erst beim erfolgreichen Beenden der Transaktion in die Datenbank synchronisiert. Im Fall des erarbeiteten Beispiels würde so ein Überführen wie in Quellcode 2.19 aussehen.

```
...
public class Haustier {
...
    em.getTransaction().begin();
    // erzeuge Max Mustermann (transient)
    Person p = new Person();
    p.setVorname("Max");
    p.setNachname("Mustermann");
    p.setGeschlecht(GESCHLECHT.MAENNLICH);
    p.setGeburtsdatum(Calendar.getInstance().getTime());

    // uebergebe an EnitityManager (managed)
    em.persist(p);

    em.getTransaction().commit(); // uebernimmt in Datenbank
    // wuerde Aenderungen und Neuanlagen verwerfen
//    em.getTransaction().rollback();
...
```

Quellcode 2.19: transient- zu managed-object

Die Transaktion kann aber auch erst im Nachhinein begonnen und sofort wieder beendet werden. Dadurch würde sich die Zeitspanne der offenen Transaktion auf ein Minimum reduzieren.

Allerdings besagt die Spezifikation nicht wann in die Datenbank geschrieben werden muss. Es bleibt dem Provider also freigestellt bereits vor dem commit()-Befehl in die Datenbank zu schreiben. Im Falle eines Rollbacks würde es in der Hand des Providers liegen die fälschlicherweise erzeugten Daten wieder aufzuräumen. Dieses Vorgehen kann somit unnötigen Aufwand verursachen, darum sind die Provider immer bemüht

[19] Wenn ein POJO geändert wurde, aber noch nicht in die Datenbank synchronisiert wurde

die Daten möglichst spät – im besten Fall erst bei Aufruf des commit()-Befehls – in die Datenbank zu schreiben.

Es ist allerdings möglich das verfrühte Schreiben in die Datenbank zu erzwingen. Dafür gibt es am EntityManager die Methode **flush()**. Dieses Vorgehen kann sinnvoll sein, falls man eine große Menge an Daten schreiben will und nicht davon ausgeht das es zu einem Rollback kommt. Der Rollback-Befehl würde somit nur die Exit-Strategie darstellen und man würde somit auch den möglichen Mehraufwand in Kauf nehmen.

Synchronisation von managed-objects (vgl. [MW12], S. 62)

Wie bereits oben erwähnt besagt der Begriff „dirty“, ob ein Objekt seit seinem Auslesen verändert wurde. Beim commit()-Befehl werden nur dirty POJOs in die Datenbank geschrieben, da von allen anderen davon ausgegangen wird, dass Sie synchron sind.

In die eine Richtung – von Java zur Datenbank – mag das wohl stimmen, allerdings ist es möglich, dass sich die Daten seitens der Datenbank durch eine andere Anwendung verändert haben. Um die POJOs mit den aktuellen Daten der Datenbank zu versorgen, gibt es die Methode **refresh()**. Allerdings sollte man diese Methode behutsam nützen, da Sie kompromisslos alle Werte – auch die bereits veränderten – mit den aktuellen Datenbank-Werten überschreibt. Um einen Synchronisationsverlust von der Datenbank ausgehend zu vermeiden sollte man die Bearbeitungszeitfenster so gering wie möglich halten. Natürlich ist dadurch nicht garantiert, dass es trotzdem zu Synchronisationsfehlern kommt. Darum sollte man – bei Multi-User-Anwendungen – die zu bearbeitenden Daten mit dem Befehl **lock()** sperren, bis die Synchronisation Richtung Datenbank abgeschlossen wurde.

Anwendung von detached objects (vgl. [MW12], S. 61)

Es gibt die Möglichkeit Objekte rein read-only aus der Datenbank auszulesen. In einem solchen Fall wird das betroffene POJO gleich nach der Abfrage vom EntityManager losgelöst und wird bei Änderung nicht vom EntityManager berücksichtigt.

```
...
    em.getTransaction().begin();
    Map<String, Object> map = new HashMap<String, Object>();
    map.put("eclipselink.read-only", Boolean.TRUE);
    // sucht Person via ID
    Person p = em.find(Person.class, new BigInteger("2"), map);
    System.out.println(p.getVorname() + " : " + p.getNachname());
    em.getTransaction().commit();
...
```

Quellcode 2.20: detached objects in JPA

Abgesehen von einem solchen Sonderfall gilt jedes aus der Datenbank geladene Objekt als managed – als ob man es mittels persist() zum EntityManager hinzugefügt hätte.

Anwendung von removed objects (vgl. [MW12], S. 63)

Sobald ein POJO mittels der Methode **remove()** am EntityManager zum Löschen markiert wurde, handelt es sich um ein „removed object“, obwohl es sowohl als POJO, als auch als Entität in der Datenbank vorhanden ist. Erst mit einem erfolgreichen commit()-Befehl wäre der Löschvorgang abgeschlossen. Solange man allerdings das POJO nicht aus der Hand lässt, kann man es mit einem erneuten persist()-Befehl einfach erneut in die Datenbank schreiben.

```
1  ...
       // die einzige Test-Person wird gesucht
3      TypedQuery<Person> tq = em.createQuery("select p from Person p", Person.class);
       Person p = tq.getSingleResult();
5
       // die Person wird geloescht ...
7      em.getTransaction().begin();
       em.remove(p);
9      em.getTransaction().commit();

11     // ... und wieder erzeugt
       em.getTransaction().begin();
13     em.persist(p);
       em.getTransaction().commit();
15 ...
```

Quellcode 2.21: removed objects in JPA

2.2 Grafische Oberfläche mittels Java FX 2

Java FX ist eine neue GUI-Technologie, die schrittweise die veraltete Variante Java Swing ablösen soll. Das erste Mal im Mai 2007 präsentiert war Java FX vorerst eine externe Technologie. Seit Java SE 6 Update 10 ist Java FX dann in den Standard aufgenommen worden und auf jedem Gerät – welches Java installiert hat – verfügbar. (vgl. [JLW14], S. 1ff)

In diesem Kapitel werden folgende Punkt erarbeitet und analysiert, die in gängigen Applikationen benötigt werden:

- **Layouting:** Es wird ein Layout gesucht, welches sich bei verschiedenen Frame-Größen sinnvoll verhält.
- **Panels/Widgets:** Es wird analysiert, ob der Aufbau eines Frames inklusive Panels auch entsprechend modular gestaltet werden kann.

- **Tabellen:** Oft werden intelligente Tabellen benötigt. Diese Tabellen sollen nicht nur alle möglichen verschiedenen Komponenten (Checkboxen, Auswahlboxen, Datumsfelder, Zahlenfelder, Textfelder) anzeigen können, sondern auch in der Lage sein die angezeigten Werte zu bearbeiten und gegebenenfalls zu sichern. Auch das Filtern soll möglich sein.
- **Buttons und Listener:** In nahezu jeder Applikation werden Buttons benötigt um eine Interaktion mit dem User zu ermöglichen. Um nach dem Drücken auf einen Button eine Aktionen ausführen zu können, muss man in der Lage sein auf einen solchen Button zu horchen. Solche Interfaces nennt man bei Java-Swing „Listener“.
- **Diagramme:** Oft werden Diagramme benötigt, um dem Anwender eine Anzeige näher bringen zu können. In Java-Swing gab es in diesem Punkt noch keine Implementierung. Entweder man hat auf externe Technologien zugegriffen oder eine eigene Lösung entwickelt. Im Gegensatz dazu bietet JavaFX bereits eine eigene Diagramm-Implementierung.

In diesem Kapitel wird schrittweise nach Lösungen und Konzepten in JavaFX für die oben genannten Punkte gesucht, um diese anschließend anhand einer Test-Applikation im nächsten Kapitel mit einer herkömmlichen Java-Swing-Lösung zu vergleichen.

2.2.1 Stage, Scenes und Nodes

In Java FX ist das, was in Java-Swing als „Frame“ bezeichnet wird – also das Fenster in dem sich die Applikation abspielt – als **Stage** bezeichnet. Während man bei Java-Swing mittels „new Frame()“ neue Fenster erzeugen konnte, wird bei Java FX alles über „Builder“ via statischer Methoden angefordert. Damit ein neues Fenster erscheint, muss die abstrakte Klasse **javafx.application.Application** vererbt werden. Mit dem Aufruf der statischen Methode **launch()** wird dann das Fenster erzeugt und automatisch die zu implementierende Methode **start()** aufgerufen. Hier muss die Implementierung der Anzeige aufgebaut werden.

```
1  ...
   public class FirstFxApplication extends Application{
3
     /**
5     * Main-Methode
      * @param args - ggf. Startparameter
7     */
     public static void main(String[]args){
9      FirstFxApplication.launch(args);
     }
11
     @Override
13   public void start(Stage primaryStage) throws Exception {
       // Implementierung muss hier eingebaut werden
15   }
   ...
```

Quellcode 2.22: erzeuge neues Fenster mittels Java FX

Das „Panel“ in Java-Swing ist in Java FX eine **Scene**. An eine „Stage“ kann nur eine „Scene“ gehängt werden, die dann etwaige Daten anzeigt. Eine „Scene“ wird über den **SceneBuilder** angefordert.

In Java FX ist es üblich diese Scenes gleich beim Erzeugen zu Konfigurieren (Layout, Größe, Nodes, ...). Diese Konfiguration läuft über Aufrufe, die alle den „SceneBuilder“ selbst als Return-Wert haben, sodass man hier beliebig viele Methoden nacheinander zum Konfigurieren aufrufen kann. Am Schluss wird die Konfiguration mit der Methode **build()** abgeschlossen.

Einzelne Komponenten werden der Scene mittels der Methode **root()** und dem Anfordern einer neuen Gruppe über den **GroupBuilder** übergeben. Hier können auch etwaige andere Nodes – die später erarbeitet werden – übergeben werden.

```
...
  @Override
  public void start(Stage primaryStage) throws Exception {
    Scene scene = SceneBuilder.create()
      .width(320)
      .height(343)
      .root(
        GroupBuilder.create()
        .children(
          // hier werden die Komponenten hinzugefuegt
        ).build()
      ).build();

    primaryStage.setScene(scene);
    primaryStage.setTitle("Beispiel");
    primaryStage.show();
  }
...
```

Quellcode 2.23: erzeuge neue Scene mittels Java FX (vgl. [JLW14], S. 22 - 26)

Komponenten die anschließend an die Scene gehängt werden, heißen in Java FX **Node**. Man kann sich also die Entwicklung eines GUI-Konzeptes mittels Java FX wie ein Theaterstück vorstellen. Erst wird die Bühne – die **Stage** – gebaut. Hier werden dann einzelne Szenen in denen interagiert wird – also **Scenes** – hinzugefügt. In diesen Szenen befinden sich Komponenten die mit dem Benutzer interagieren – die **Nodes**. (vgl. [JLW14], S. 35)

2.2.2 Komponenten und Listener

In diesem Unterkapitel werden die wichtigen Komponenten untersucht. Es wird analysiert, wie Sie erzeugt und angesprochen werden.

2.2.2.1 Text Element (vgl. [JLW14], S. 213ff)

Texte – in Java-Swing „Labels“ genannt – dienen dazu einem Benutzer die Oberfläche zu erklären. Ein typischer Anwendungsfall ist das Label neben einem Textfeld, welches besagt, was in das Textfeld geschrieben werden soll.

```
...
  @Override
  public void start(Stage primaryStage) throws Exception {

    // erzeuge einfachen Text
    Text text = TextBuilder.create()
      .text("Username:")
      .build();

    // montiere Text inmitten der Anzeige
    Scene scene = SceneBuilder.create()
      .width(100)
      .height(50)
      .root(
        BorderPaneBuilder.create()
        .center(text)
        .build()
      ).build();

    primaryStage.setScene(scene);
    primaryStage.setTitle("Testanwendung");
    primaryStage.show();
  }
...
```

Quellcode 2.24: erzeuge ein Text-Node

Man sieht in Quellcode 2.24 auf ersten Blick, dass sich die Reihenfolge beim Erstellen einer Komponente stark zu Java-Swing unterscheidet. Während man bei Java-Swing erst ein Objekt erzeugt, um es anschließend zu konfigurieren, wird hier erst eine Konfiguration erstellt aus der heraus man das Objekt erzeugt.

Zusätzlich zu dieser neuen Erzeugungsart kann man allerdings auch auf herkömmlichem Weg erst ein Objekt erzeugen und dann konfigurieren. Auf diese Variante wird hier nicht eingegangen, da Sie kein neues Feature darstellt.

Diese Vorgehensweise hat den Vorteil, dass man aus einer Konfiguration heraus mehrere Objekte erzeugen kann. Bei Java-Swing müsste man in einem solchen Bedarfsfall die Konfiguration am jeweiligen Objekt iterieren. Wie ein solcher Text in Java FX aussieht, wir in Abbildung 2.2 dargestellt.

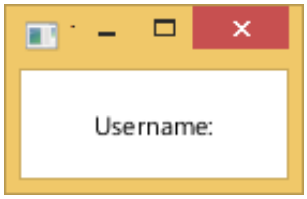


Abbildung 2.2: Text mit Java FX

Eine andere wichtige Eigenschaft ist das Manipulieren dieser Texte. Während man bei Java-Swing lediglich am Objekt selbst die Methode „setText()“ aufgerufen hat, gibt es bei Java FX hierfür verschiedene Property-Objekte. Außerdem kann ausgewählt werden, ob das Property unidirektional – ohne Rückmeldung (falls der User mit der Komponente interagiert, damit der gesetzte Wert nicht verloren geht) – oder bidirektional angebunden wird.

```
...
    StringProperty sp = new SimpleStringProperty();
    // Property wird unidirektional angebunden
    text.textProperty().bind(sp);
    sp.setValue("Neuer Username:");
...
```

Quellcode 2.25: Textmanipulation mittels Property

Zusätzlich zu dieser neuen Variante kann man die Komponenten allerdings auch wie in Java-Swing via Getter- und Setter-Methoden direkt an der Komponente manipulieren. Hier gibt es viele verschiedene Properties wie zum Beispiel

- StringProperty
- DoubleProperty
- IntegerProperty

Somit kann der Entwickler durch die Verwendung dieser Properties einem normalen Text seinen Inhalt vorschreiben. In Java-Swing müsste man für eine solche Funktionalität eine eigene abgeleitete Klasse erstellen.

2.2.2.2 Textfield Element (vgl. [JLW14], S. 213ff)

Auch hier wird erst konfiguriert und anschließend erzeugt. Abgesehen davon ist das Textfeld relativ ähnlich zu dem aus Java-Swing. Es bietet allerdings wesentlich mehr Konfigurationsmöglichkeiten.

Abgesehen von der erweiterten Konfigurierbarkeit wird hier für alle möglichen Listener-Funktionalitäten immer der **ChangeListener** verwendet. Dieser Handler stellt einen universellen Listener dar, der je nach Montagepunkt zu verschiedenen Zeitpunkten aufgerufen wird.

In Java-Swing gab es zum Beispiel für die Fokussierung einer Komponente den „Focus-Listener“ und für das Drücken einer Taste im ausgewählten Textfeld den „KeyListener“. In Java FX hingegen gibt es nur den „ChangeListener“ der hierfür entweder am „focusedProperty“ oder am „textProperty“ angehängt wird.

```
...
    // erzeugen eines Textfeldes
    final TextField tf = TextFieldBuilder.create()
      .promptText("Username")
      .prefColumnCount(16)
      .focusTraversable(false)
      .build();

    // ein universeller Listener
    ChangeListener listener = new ChangeListener() {
      @Override
      public void changed(ObservableValue observable, Object oldValue,
          Object newValue) {
        System.out.println("neuer Text: " + tf.getText());
      }
    };

    // entspricht KeyListener aus Java Swing
    tf.textProperty().addListener(listener);
    // entspricht FocusListener aus Java Swing
    tf.focusedProperty().addListener(listener);
...
```

Quellcode 2.26: erzeuge ein Textfeld-Node mit Listener

Ausgehend von Quellcode 2.26 entsteht in Java-FX die Anzeige aus Abbildung 2.3 mit transparenter Beschreibung (Username).

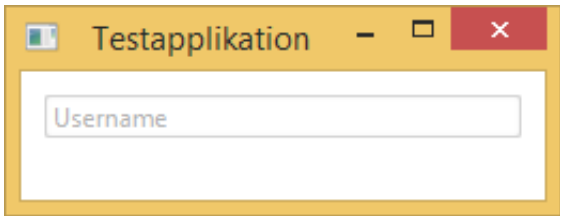


Abbildung 2.3: Textfeld mit Java FX

2.2.2.3 Checkbox Element (vgl. [JLW14], S. 213ff)

Auch eine Checkbox wird erst über einen Builder definiert und anschließend erzeugt. Hier fällt sofort die erste Schwachstelle von Java FX auf: Genau wie beim Textfeld kann auch hier ein „ChangeListener“ angehängt werden. Allerdings wird dieser Listener niemals aufgerufen. Es ist nicht gut Methoden zur Verfügung zu stellen, die nicht verwendbar sind. Um trotzdem auf eine Aktion der Checkbox horchen zu können gibt es den **EventHandler**.

```
...
    CheckBox cb = CheckBoxBuilder.create()
        .text("CheckBox")
        .build();

    cb.onActionProperty().set(new EventHandler<ActionEvent>() {
      @Override
      public void handle(ActionEvent event) {
        System.out.println(event.getEventType() + " : " +
          ((CheckBox)event.getSource()).isSelected());
      }
    });

    // verwirrende Moeglichkeit einen Listener hinzuzufuegen
    cb.onActionProperty().addListener(new ChangeListener() {
      @Override
      public void changed(ObservableValue observable, Object oldValue,
          Object newValue) {
        // wird niemals aufgerufen
        System.out.println("Listener: " + newValue);
      }
    });

    \\
    BooleanProperty pb = new SimpleBooleanProperty();
    cb.selectedProperty().bindBidirectional(pb);
...
```

Quellcode 2.27: erzeuge eine Checkbox-Node mit Handler

Wie der Quellcode 2.27 zeigt, kann auch bei einer Checkbox die Kommunikation über ein Property gesteuert werden. Auch hier gibt es die Möglichkeiten es unidirektional oder bidirektional anzubinden. Eine Checkbox in Java FX sieht wie in Abbildung 2.4 dargestellt aus.

Abbildung 2.4: Checkbox mit Java-FX

2.2.2.4 ChoiceBox Element (vgl. [JLW14], S. 213ff)

Eine **ChoiceBox** ist das, was in Java-Swing eine „ComboBox“ ist – also ein Auswahlfeld. Im Gegensatz zur Checkbox(boolean) und zum Textfeld(String), ist bei einem Auswahlfeld niemals im Vorhinein klar mit welcher Art von Objekt es befüllt wird. Darum gibt es genau wie in Java-Swing auch bei Java-FX die Möglichkeit bereits beim Erzeugen des Feldes mittels Generics zu beschreiben, mit welcher Art von Objekt das Auswahlfeld befüllt wird.

Daraus folgen viele Vorteile wie zum Beispiel, dass der „ChangeListener“ – der auch

hier verwendet wird – in diesem Fall bereits den Objekttyp des Inhaltes kennt und die Attribute „oldValue“ und „newValue“ nicht mehr gecastet werden müssen.

```
...
    // Liste mit Werten fuer das Feld
    final ObservableList<String> list = FXCollections.observableArrayList(
        "Item1", "Item2", "Item3");

    // Erzeugen des Feldes
    ChoiceBox<String> cb = ChoiceBoxBuilder.<String>create()
        .items(list)
        .prefWidth(100)
        .build();

    // Listener kann entweder an "selectedItem..." oder "selectedIndex..." gehaengt
         werden
    cb.getSelectionModel().selectedItemProperty().addListener(new ChangeListener<
        String>() {
      @Override
      public void changed(ObservableValue<? extends String> observable,
          String oldValue, String newValue) {
        System.out.println("selected: " + newValue);
      }
    });

    // Hinzufuegen und Loeschen von Items
    list.add("item8");
    list.remove(1);
...
```

Quellcode 2.28: erzeuge eine ChoiceBox-Node mit Listener

Bei der „ChoiceBox“ ist die Kommunikation über ein Property nicht notwendig, da es hierfür die **ObservableList** gibt, welche die Items zur Verfügung stellt. Sobald in diese Liste Items mittels „add()“ hinzugefügt oder mittels „remove()“ entfernt werden, passt sich das Auswahlfeld in Echtzeit an die Änderungen an, ohne manuell aktualisiert werden zu müssen. Ein Auswahlfeld sieht in Java-FX wie in Abbildung 2.5 beschrieben aus.

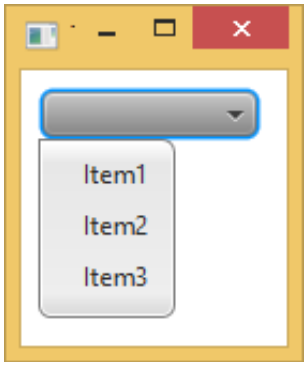


Abbildung 2.5: ChoiceBox mit Java-FX

2.2.2.5 Button Element (vgl. [JLW14], S. 213ff)

Genau wie bei der „CheckBox“ kann man beim „Button“ einen Listener anhängen, der niemals zum Einsatz kommt. Auch hier wird mittels eines **EventHandler** gehorcht. Abgesehen davon, dass es sich um einen „Button“ handelt, ist die Konfiguration identisch zu der von der „CheckBox“.

```
...
    Button btn = ButtonBuilder.create()
      .text("fertig")
      .prefWidth(100)
      .build();

    btn.onActionProperty().set(new EventHandler<ActionEvent>() {
      @Override
      public void handle(ActionEvent event) {
        System.out.println(event.getEventType() + ": " +
            ((Button)event.getSource()).getText());
      }
    });
...
```

Quellcode 2.29: erzeuge eine Button-Node mit Handler

Ein solches Feld sieht mit Java-FX wie in Abbildung 2.8 beschrieben aus.

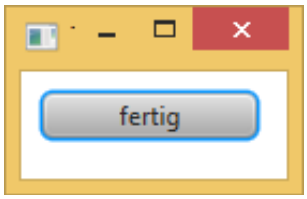


Abbildung 2.6: Button mit Java-FX

2.2.3 Tabellen in Java FX

Tabellen spielen in der Oberfläche einer komplexen Applikation immer eine große Rolle. Hier kann man nicht nur verschiedenste Werte anzeigen, sondern die Tabelle auch so gestalten, dass zu verschiedenen Zeitpunkten einzelne Zellen auf komplett eigene Art und Weise editierbar sind.

In Java-Swing waren die Möglichkeiten – wenn es um Tabellenkonfiguration geht – sehr vielfältig. Man konnte die Anzeige jeder Zelle durch Renderer bis ins kleinste Detail individualisieren. Auch wenn es um das Editieren geht gab es hier die Möglichkeit verschiedenste Editoren zu erstellen, die sich immer nach den Wünschen des Kunden anpassen ließen.

Da Tabellen ein Kernthema sind, werden hier die oben genannten Punkte in Java FX genauer analysiert und erarbeitet.

Es wird schrittweise für die gängigen Datentypen „String“, „Date“, „Double“, „Boolean“ und „Enumeration“ nach Spaltendefinitionen gesucht, mit denen die Werte angezeigt und manipuliert werden können.

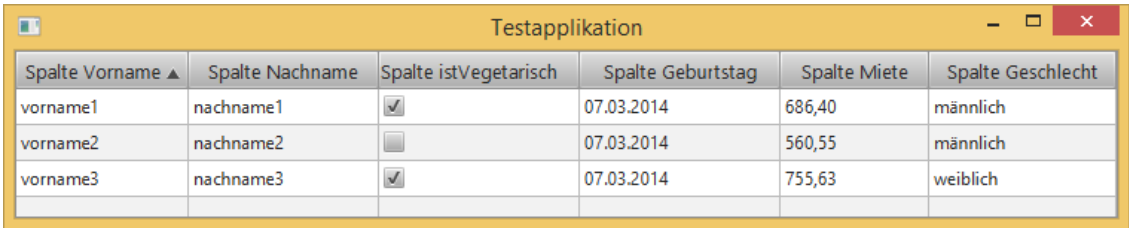


Abbildung 2.7: Java FX TableView mit verschiedenen Datentypen

2.2.3.1 Tabellenaufbau (vgl. [JLW14], S. 205ff)

Während in Java-Swing der Inhalt einer Tabelle über das „JTableModel“ beschrieben wird, welches anschließend an ein „JTable“ übergeben wird fällt in Java-FX diese Unterteilung weg. In Java-Swing war es äußerst mühsam mit den Standardkomponenten klarzukommen, darum ist es üblich sich selbst eine Art Framework-Komponente zu schreiben die einem die Erstellung von komplexen Tabellen erleichtern. In Java FX hingegen gibt es bereits so ein Konfigurationstool. Auch hier gibt es einen Builder – den **TableViewBuilder**, welcher mit einer Liste aus POJOs befüllt wird. Die POJO-Klasse wird als Generic bei der Erzeugung übergeben. Anschließend müssen im Builder nur noch die einzelnen Spalten konfiguriert werden und die Quell-Tabelle übergeben werden.

Quellcode 2.30 beschreibt ein Beispiel wie eine Tabelle mit POJOs der Klasse Person befüllt wird. Eine Person besteht in diesem Fall aus „Vornamen“, “Nachnamen“, „Mietkosten“, „Geburtsdatum“, „istVegetarisch“ und dem Property „Geschlecht“.

```
...
TableView<Person> tableView = TableViewBuilder.<Person>create()
        .columns(
          //hier werden die Spalten definiert
        )
        .items(
          // hier wird die Liste mit Daten uebergeben
          // Die Liste wird automatisch upgedatet falls in
          // der Oberflaeche Aenderungen vorgenommen werden.
        )
        .editable(true) // beschreibt ob Tabelle editierbar ist
        .build();
...
```

Quellcode 2.30: erzeuge eine TextView-Node

Zusätzlich gibt es in Java-Swing zur Manipulation der Anzeige je Spalte einen „Renderer“. Zur Manipulation des Eingabefeldes – falls man die Spalte editieren will – gab es hier „Editors“. Diese mussten mit viel Aufwand erzeugt und an der richtigen Stelle verwendet werden. In Java-FX hingegen werden auch diese beiden Punkte einfach bei der Spaltenerzeugung definiert.

Das Erzeugen von komplexen Tabellen ist zwar sehr gut gestaltet, allerdings noch schwach dokumentiert und fehleranfällig. Auch sind bei der Implementierung verschiedenster Spaltentypen richtige Bugs aufgefallen, die bei der Analyse sehr viel Zeit in Anspruch genommen haben.

2.2.3.2 Text-Spalten (vgl. [JLW14], S. 205ff)

Textspalten sind relativ einfach zu implementieren, da sie den Standard einer Spalte darstellen. Bei Java FX wird die Kommunikation mit den Properties im dazugehörigen POJO über die Klasse **StringProperty** gesteuert. Diese muss mit einem Namen – über den Sie anschließend bei der Tabellenkonfiguration angesprochen wird – identifiziert.

```
1  ...
   public class Person {
3
     /**
5     * Vorname StringProperty
      */
7    private StringProperty vornameProperty;

9    /**
      * gibt das Vorname-StringProperty zurueck
11    * falls Property null - wird es mit dem Namen "vorname" erzeugt
      * @return - vornameProperty
13    */
     public StringProperty getVornameStringProperty(){
15     if(vornameProperty == null)
         vornameProperty = new SimpleStringProperty(this, "vorname");
17     return vornameProperty;
     }
19
     /**
21    * erlaubt das Setzen eines Vornamens
      * @param vorname - der zu setzende Vorname
23    */
     public void setVorname(String vorname){
25     getVornameStringProperty().set(vorname);
     }
27
     /**
29    * liest einen Vornamen aus dem StringProperty aus
      * @return der Vorname als String
31    */
     public String getVorname(){
33     return getVornameStringProperty().get();
     }
35 ...
```

Quellcode 2.31: ein StringProperty im Tabellen-POJO

Das Property aus Quellcode 2.31 wird anschließend über seinen Namen „vorname“ angesprochen. Um die Spalte in der Anzeige richtig darzustellen, muss Sie durch den **TableColumnBuilder** erzeugt werden. Hier müssen sowohl der Typ des POJOs als auch der Typ der Spaltenklasse als Generic übergeben werden. Anschließend wird der Spaltenname für den Header übergeben und die Anzeige über eine **ValueFactory** – der der Name des dazugehörigen Properties im POJO übergeben wird – definiert.

```
...
TableColumn<Person, String> vornameSpalte = TableColumnBuilder.<Person, String>
    create()
  .text("Spalte Vorname")
  .cellValueFactory(new PropertyValueFactory<Person, String>("vorname"))
  ...
  .build();
...
```

Quellcode 2.32: erzeugen einer Textspalte im TableView

Nach diesem Schritt aus Quellcode 2.32 funktioniert zwar die Anzeige, das Editieren ist aber noch nicht möglich. Dazu muss man ein **Callback** definieren. Hier tritt bereits der erste Fehler in Java-FX auf.

Während bei anderen Spalten das Definieren eines „Callbacks“ ausreicht, wird bei Textfeldern der Wert nicht automatisch an das Property im POJO übergeben. Es wird von einem Fehler ausgegangen, da dieser Vorgang bei allen anderen Spaltentypen reibungslosen funktioniert. Damit auch bei Textfeldern der Wert in das POJO geschrieben wird, muss wie in Quellcode 2.33 beschrieben beim Verlassen der Zelle der Wert manuell übernommen werden.

```
...
.cellFactory(new Callback<TableColumn<Person,String>, TableCell<Person,String>>() {
  @Override
  public TableCell<Person, String> call(TableColumn<Person, String> param) {
    return new TextFieldTableCell<Person, String>(){{
      setConverter(new StringConverter<String>() {
        @Override
        public String fromString(String string) {
          return string;
        }
        @Override
        public String toString(String object) {
          // damit die Zelle mit ENTER verlassen werden kann
          return object.replace("\n", "");
        }
      });
    }};
  }
})
// nur bei Textfeldern notwendig: manuelles Speichern ins POJO
.onEditCommit(new EventHandler<TableColumn.CellEditEvent<Person, String>>() {
  @Override
  public void handle(CellEditEvent<Person, String> event) {
    TableColumn<Person, String> col =
        ((TableColumn<Person, String>)event.getSource());
    Person p = (Person)col.getTableView().getItems()
        .get(event.getTablePosition().getRow());
    p.setVorname(event.getNewValue());
  }
})
...
```

Quellcode 2.33: Textfeld-Spalte im TableView editierbar gestalten

2.2.3.3 Zahlen-Spalten

Zahlenspalten sind nichts anderes als Textspalten deren Wert vor dem Schreiben ins dazugehörige **DoubleProperty** gecastet werden muss. Das Property ist – abgesehen vom Typ und dem Namen „miete“ – ident zum Textfeld-Property.

Auch die Tabellendefinition unterscheidet sich bis auf die Generic-Übergabe „<Person, Double>“ kaum. Lediglich der **StringConverter** muss ausgetauscht werden, damit im POJO Double-Werte ankommen.

```
...
new StringConverter<Double>() {
  private DecimalFormat format = new DecimalFormat("#,###,###,##0.00" );

  @Override
  public Double fromString(String string) {
    try {
      return format.parse(string).doubleValue();
    } catch (ParseException e) {
      e.printStackTrace();
    }
    return null;
  }

  @Override
  public String toString(Double object) {
    return format.format(object);
  }
}
...
```

Quellcode 2.34: Zahlen-Spalte im TableView konvertieren

2.2.3.4 Datum-Spalten

Hier kommt es zum nächsten Mangel in Java-FX. Obwohl Datumseingaben in Tabellen keine Seltenheit darstellen, gibt es bei Java-FX kein Datums-Property. Daher muss man sich selbst um die Objektverwaltung kümmern. Als universelles Property muss das **ObjectProperty<Date>** mit dem Generic „Date“ verwendet werden. Abgesehen davon unterscheidet sich das Property nicht von den Anderen.

Im „TableView“ wird das Datumsfeld wie ein Textfeld implementiert – lediglich mit dem Generic „<Person, Date>“. Auch hier muss nur der Converter angepasst werden, damit im POJO wieder ein „Date“-Wert ankommt.

```
...
new StringConverter<Date>() {
  SimpleDateFormat format = new SimpleDateFormat("dd.MM.yyyy");
  @Override
  public Date fromString(String string) {
    try {
      return format.parse(string);
    } catch (ParseException e) {
      e.printStackTrace();
```

```
10      }
        return null;
12    }

14    @Override
      public String toString(Date object) {
16      return format.format(object);
      }
18  }
...
```

Quellcode 2.35: Datum-Spalte im TableView konvertieren

2.2.3.5 Checkbox-Spalten

Diese Spalten haben bei der Analyse die meisten Probleme gemacht. Es gibt zwar die vorgefertigte CellFactory **CheckBoxTableCell**, allerdings übernimmt diese standardmäßig nicht den gesetzten Boolean Wert aus dem Property.

Der Fehler – der in keinem begutachteten Artikel dokumentiert war und auch bei keinem anderen Spaltentyp vorkommt – entsteht durch die Getter-Methode des Properties im POJO. Die Methoden werden nach dem Java Standard immer mit den Worten „get“ und „set“ begonnen. Wenn man sich an diese Konvention hält, ist die CheckBox in der Tabelle niemals ausgewählt, obwohl im POJO das dazugehörige Property „true“ gesetzt ist. Wenn man allerdings die Getter-Methode auf das Property genauso benennt wie das Property selbst, funktioniert die Tabelle wundersamerweise.

```
1   ...
    /**
3    * istVegetarisch BooleanProperty
     */
5   private BooleanProperty vegetarianProperty;

7   /**
     * Methode erstellt und gibt ein BooleanProperty zurueck
9    *  mit "get" im Methodennamen - also "getVegetarianProperty
     *  funktioniert es nicht mehr!
11   * @return Property istVegetarisch
     */
13  public BooleanProperty vegetarianProperty(){
      if(vegetarianProperty == null)
15      vegetarianProperty = new SimpleBooleanProperty(this, "vegetarian", false);
      return vegetarianProperty;
17  }
...
```

Quellcode 2.36: BooleanProperty im TableView POJO

Die Lösung aus Quellcode 2.36 für das Problem ist durch einen vermuteten Fehler in der Reflection-Funktionalität der „CheckBoxTableCell“ zustande gekommen. Ob der Fehler wirklich dort liegt, konnte nicht ermittelt werden. Die dazugehörige Spalte im TableView wird wie in Quellcode 2.37 definiert.

```
...
TableColumn<Person, Boolean> istVegetarischSpalte = TableColumnBuilder.<Person,
    Boolean>create()
  .text("Spalte istVegetarisch")
  .cellValueFactory(new PropertyValueFactory<Person, Boolean>("vegetarian"))
  .cellFactory(new Callback<TableColumn<Person,Boolean>,
      TableCell<Person,Boolean>>() {
    @Override
    public CheckBoxTableCell<Person, Boolean> call(
        TableColumn<Person, Boolean> param) {
      return new CheckBoxTableCell<Person, Boolean>();
    }
  })
  .build();
...
```

Quellcode 2.37: CheckBox-Spalte im TableView konvertieren (vgl. [Mar12])

Wie man anhand Quellcode 2.37 sofort sieht, ist diese Spaltendefinition um einiges kürzer als die im Textfeld. Das folgt daraus, dass man sich hier die Implementierung eines „Callbacks“ spart, da die Kommunikation mit dem Property direkt – ohne eigene Entwicklung – funktioniert.

2.2.3.6 Auswahl-Spalten

Bei einer Auswahlbox würde es sich anbieten die auszuwählenden Objekte in einer Enumeration zu definieren. Wie Quellcode 2.38 zeigt muss auch hier wieder auf das universelle Property **ObjectProperty** zurückgegriffen werden.

```
...
  /**
   * beschreibt die moeglichen Geschlechter
   */
  public enum GESCHLECHT{maennlich, weiblich};

  /**
   * Property haelt das aktuell gewaehlte Geschlecht
   */
  private ObjectProperty<GESCHLECHT> geschlechtProperty;
...
```

Quellcode 2.38: ObjectProperty mit enum-Generic

Hier funktioniert die Implementierung im „TableView“ einfach und fehlerfrei. Man muss lediglich durch den **ComboBoxTableCellBuilder** den Wertebereich der Box definieren und das Objekt setzen.

Das Einzige was hier negativ auffällt ist die unterschiedliche Namensgebung zum normalen Auswahlfeld. Während die Auswahl-Node „ChoiceBox“ heißt, ist der Name des Auswahlfeldes in der Tabelle „ComboBox“. In Java-Swing wurden die Namen bei solchen Komponenten gleich gehalten. Die Implementierung einer solchen Spalte sieht im „TableView“ wie in Quellocde 2.39 beschrieben aus.

```
...
TableColumnCell<Person, GESCHLECHT> geschlechtSpalte =
  TableColumnBuilder.<Person, GESCHLECHT>create()
  .text("Spalte Geschlecht")
  .cellValueFactory(new PropertyValueFactory<Person, GESCHLECHT>("geschlecht"))
  .cellFactory(new Callback<TableColumn<Person,GESCHLECHT>,
      TableCell<Person,GESCHLECHT>>() {
    @Override
    public TableCell<Person, GESCHLECHT> call(TableColumn<Person,
        GESCHLECHT> arg0) {
      return ComboBoxTableCellBuilder.<Person, GESCHLECHT>create()
        .items(GESCHLECHT.values())
        .build();
    }
  }).build();
...
```

Quellcode 2.39: CheckBox-Spalte im TableView konvertieren

2.2.4 Diagramme in Java FX

Oft werden Diagramme in Applikationen dazu verwendet dem Benutzer die angezeigten Zahlenwerte näher zu bringen und visuell zu vergleichen. Java-Swing bietet hierfür keine Implementierung an. Oft wird daher auf OpenSource-Technologien wie **JFreeChart** zurückgegriffen, die allerdings mittlerweile veraltet und in ihrer Konfigurierbarkeit eingeschränkt sind.

Java FX bietet hingegen eine moderne Diagramm-Technologie an. Abgesehen davon sind Charts hier als „Nodes“ definiert und können somit leicht in der Oberfläche eingebunden werden.. Java FX unterscheidet zwischen null[20]- und zwei-achsigen Diagrammen. In diesem Kapitel werden die zweiachsigen Balkendiagramme in Java-FX analysiert. (vgl. [JLW14], S. 307f)

2.2.4.1 Diagramm-Werte (vgl. [JLW14], S. 323)

Die Daten bei Balkendiagrammen in Java FX werden durch **Series** im Key/Value-Prinzip erfasst. Diese Series müssen einen „String“-Key besitzen. Dieser Key beschreibt die Position des Balkens an der X-Achse. Der Value-Wert stellt die Höhe des Balkens entlang der Y-Achse dar und sollte hingegen vom Typ „Number“ sein. Wie bei den meisten Objekten in Java-FX sollte man auch hier die Generics für diese Formation übergeben.

Aus einer „Serie“ werden immer alle Spalten in der selben Farbe dargestellt. Die fertigen „Series“ werden anschließend – wie so oft in Java FX – an eine **ObservedList** übergeben. Wichtig wäre es auch noch einer „Serie“ einen Namen zu geben, der anschließend in der Legende verwendet wird.

[20] zum Beispiel Kreisdiagramme

```
...
  Series<String, Number> einnahmen = new Series<String, Number>();
  einnahmen.setName("Einnahmen");
  einnahmen.getData().add(
    new XYChart.Data<String, Number>("2011", (int)(Math.random()*1000)));
...
  Series<String, Number> ausgaben = new Series<String, Number>();
  ausgaben.setName("Ausgaben");
  ausgaben.getData().add(
    new XYChart.Data<String, Number>("2011", (int)(Math.random()*1000)));
...
  ObservableList<XYChart.Series<String, Number>> list =
    FXCollections.observableArrayList();
  list.add(einnahmen);
  list.add(ausgaben);
...
```

Quellcode 2.40: Erzeuge Series für ein BarChart

2.2.4.2 Diagramm-Achsen

Um die Achsen zu definieren, muss man jeweils ein Objekt erzeugen, welchen den Inhalt der Achse beschreibt – in diesem Fall **CategoryAxis** und **NumberAxis**. Diese beiden Achsendefinitionen beschreiben „String“-Kategorisierung und „Number“-Werte. Bei der Übergabe an das „BarChart“-Objekt wird diese Abbildung mittels Generics überprüft.

Abgesehen davon kann man an der Achse mehrere Konfigurationen vornehmen – wie zum Beispiel die Beschriftung der Achse. Für den Normalfall reicht es allerdings vollkommen ein frisch erzeugtes, nicht konfiguriertes Objekt zu übergeben.

2.2.4.3 Balkendiagramm erzeugen

Auch hier treten keine Probleme auf. Das Erzeugen läuft einfach und reibungslos. Es werden dem Konstruktor die gewählten Achsendefinitionen übergeben. Es sollten auch die gewählten Objekttypen als Generics übergeben werden. In einem solchen Fall erkennt der Compiler sofort, falls die Achsendefinitionen falsch gewählt wurden. Anschließend muss noch die Liste mit den „Series“ übergeben werden. Optional wäre es sinnvoll einen Namen zu setzen. Danach kann das Diagramm bereits auf die Oberfläche montiert werden.

```
...
  BarChart<String, Number> barChart =
    new BarChart<String, Number>(
      new CategoryAxis(), new NumberAxis());
  barChart.setData(list);
  barChart.setTitle("Werte");
...
```

Quellcode 2.41: erzeuge Series für ein BarChart

Somit kann relativ rasch und ohne Probleme ein sinnvolles Diagramm erstellt werden. Hier scheint Java-FX bereits ausgereift zu sein. Aus diesem kurzen Code entstand das anschauliches Diagramm in Abbildung 2.8.

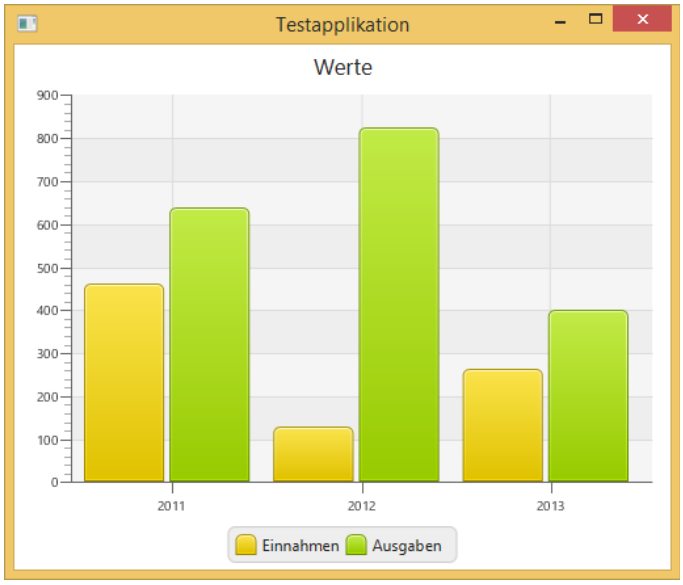


Abbildung 2.8: Balkendiagramm mittels Java-FX

2.2.5 Layouting in Java FX

Das dynamische Layouting spielt in modernen Applikationen eine wichtige Rolle. Nur selten ist es der Fall, dass eine Anzeige sich nie verändert und daher statisch definiert werden kann. Oft muss die Anzeige einer Applikation in verschiedenen Fenstergrößen funktionieren, aber auch der Inhalt der Anzeige kann stark variieren. Zum Beispiel kann man nie vorhersagen wie sich der Inhalt einer Tabelle in einer Applikation entwickeln wird. Man muss darauf gefasst sein, dass die Tabelle über die Fenstergröße hinausläuft aber trotzdem nicht die darunter befindlichen Komplementen verdrängt. Hier kann zum Beispiel durch Scroll-Balken Abhilfe geleistet werden. Oft ist die Lösung allerdings nicht so einfach. Darum ist es wichtig, dass eine GUI-Technologie sinnvolles Layouting ermöglicht.

In Java-Swing waren folgende vier Layoutings die am Meisten verwendeten:

- **FlowLayout** war ein Layout, welches die enthaltenen Komponenten einfach nebeneinander darstellte. Falls die Zeile bereits voll war, wurde die folgende Komponente einfach in eine neue Zeile verschoben. Das Layouting ist zwar dynamisch und sorgt dafür, dass keine Komponente außerhalb des Fensters verschwindet, dieses sieht allerdings oft sehr unaufgeräumt aus und kann daher unübersichtlich sein.
- **BorderLayout** ist ein Layout, welches das Fenster in fünf Bereiche teilt: NORTH,

EAST, SOUTH, WEST und CENTER. Während die vier Himmelsrichtungen statische Flächen darstellen – die sich in ihrer Größe unabhängig der Fenstergröße nicht verändern – wird die Komponente im CENTER abhängig von der Fenstergröße in die Länge und Breite vergrößert und verkleinert.

- **GridLayout** ist ein Tabellenlayout. Hier wird bei der Erzeugung mitgegeben, wie viele Spalten die Tabelle enthalten soll. Die Zeilenanzahl ergibt sich aus der Anzahl der übergebenen Komponenten. Dieses Layout ist zwar in einer Tabellenstruktur aufgebaut, stellt aber keine Tabelle dar. Es wird sehr oft verwendet um aufbereitete Daten darzustellen.
- **SpringLayout** ist ein Layout, in dem die Positionen der Komponenten relativ zu einander übergeben wird. Dieses Layouting erfordert zwar viel Code in der Erstellung, erlaubt aber das beste dynamische Layouting. Bei jeder Komponente wird übergeben, in welcher Himmelsrichtung (NORTH, EAST, SOUTH, WEST) es neben welcher anderen Komponente liegt. Die Nachbarskomponente kann allerdings auch das Fenster selbst sein.

Für vier dieser Layoutingarten soll ein sinnvolles Java FX Pendant gesucht und analysiert werden.

2.2.5.1 FlowPane, VBox und HBox in Java FX (vgl. [Org14])

In Java FX gibt es drei verschiedene Layoutings, welche ein einfaches, dynamisches Befüllen der Oberfläche – ähnlich wie beim Java-Swing „FlowLayout“ – ermöglichen. Diese unterscheiden sich lediglich darin in welche Richtung die Anzeige weiter befüllt wird.

FlowPane

Genau wie bei Java-Swing wird hier eine Komponente neben der anderen angehängt. Sobald die Zeile voll ist, wird einfach die Nächste benutzt. Das Erzeugen einer solchen „Node“ erfolgt auch hier über einen Builder – den **FlowPaneBuilder**.

Hier werden einfach die „Children“-Elemente nacheinander übergeben. Die Implementierung ist analog zu Java-Swing sehr einfach und problemlos. Der Nachteil, dass die Anzeige möglicherweise unübersichtlich wird, ist auch identisch.

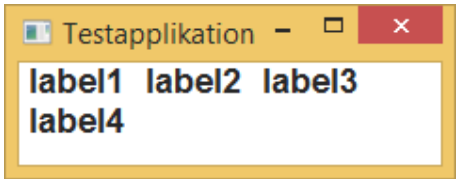


Abbildung 2.9: FlowPane in Java FX

HBox

Eine **HBox** hingegen breitet sich lediglich horizontal aus. Falls der Fensterrand erreicht ist, werden alle enthaltenen Komponenten verschmälert. Es wird keine Komponente hinter den Fensterrand verschoben. Die Implementierung erfolgt über den **HBoxBuilder** und ist ansonsten analog zum „FlowPane“.

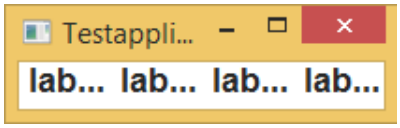


Abbildung 2.10: HBox in Java FX

VBox

Eine **VBox** ist das vertikale Pendant zur „HBox“. Im Gegensatz zur „HBox“ verschwinden hier Komponenten außerhalb des Fensters, anstatt alle Anderen zu verschmälern. Das ist ein äußerst sinnvolles Vorgehen, da ein vertikales Verschmälern zur gänzlichen Unlesbarkeit – und somit zur Unverständlichkeit – der übrigen Komponenten führen würde. Auch hier gibt es einen Builder – den **VBoxBuilder**.

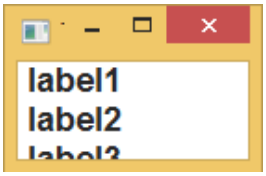


Abbildung 2.11: VBox in Java FX

2.2.5.2 BorderPane in Java FX (vgl. [Org14])

In Java FX gibt es glücklicherweise einen direkten Nachkommen zum oft verwendeten „BorderLayout“ – das **BorderPane** – welches über den **BorderPaneBuilder** erzeugt wird. Im Gegensatz zum Java-Swing Panel wird hier allerdings die Positionierung über TOP, RIGHT, BOTTOM, LEFT und CENTER definiert. Ansonsten verhält sich die Oberfläche wie aus Java-Swing gewohnt.

Man sollte einem dieser Orte im „BorderPane“ allerdings nicht direkt eine Anzeige-Komponente übergeben, sondern noch eine Panel-Node, wie zum Beispiel eine „HBox“, die nur ein Element enthält. Somit bleibt der Ort im „BorderPane“ besser konfigurierbar, was das Alignment und andere Eigenschaften betrifft.

```
...
BorderPane borderPane = BorderPaneBuilder.create()
  .padding(new Insets(10, 10, 10, 10))
  .top(
    HBoxBuilder.create()
    .alignment(Pos.CENTER)
    .children(labelTop)
    .build()
  )
  .right(
    HBoxBuilder.create()
    .alignment(Pos.CENTER)
    .children(labelRight)
    .build()
  )
  ...
  ).build();
...
```

Quellcode 2.42: erzeuge BorderPane in Java FX

Auch bei BorderPanes profitiert man in Java FX von der einfache Konfigurierbarkeit des Builders. Die Anzeige des „BorderPane“ verhält sich bei verschiedenen Fenstergrößen – wie aus Java-Swing gewohnt – dynamisch.

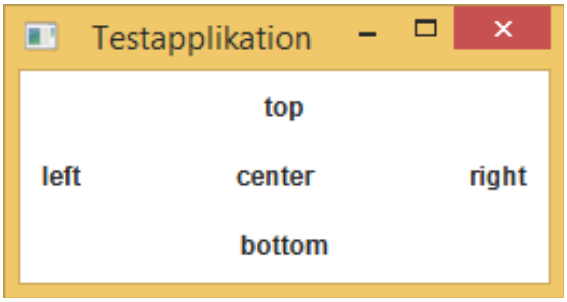


Abbildung 2.12: BorderPane in Java FX

2.2.5.3 GridPane in Java FX (vgl. [Org14])

Ein einfaches Tabellen-Panel wird über das **GridPane** dargestellt – welches gewohnt einfach über den **GridPaneBuilder** konfiguriert werden kann. Im Gegensatz zu Java-Swing erfolgt die Positionierung der einzelnen Komponenten nicht durch die definierte Spaltenanzahl – und einem damit eingehenden automatischen Zeilenumbruch sobald mehr Elemente als definierte Spalten hinzugefügt werden sondern durch Koordinaten-übergabe. Erst von dieser Positionierung ausgehend wird klar, wie groß die Tabelle wirklich ist. Diese Implementierungsart hat im Gegensatz zu Java-Swing vor allem den Vorteil der Übersichtlichkeit im Code. Man erkennt sofort auf einen Blick, wo welche Komponente liegen sollte.

```
...
GridPane content = GridPaneBuilder.create()
  .vgap(5) // Abstand zwischen Zeilen
  .hgap(30) // Abstand zwischen Spalten
  .build();

content.add(label1, 0, 0);
content.add(label2, 1, 0);
content.add(label3, 0, 1);
content.add(label4, 1, 1);
...
```

Quellcode 2.43: erzeuge GridPane in Java FX

Die Anzeige im Quellcode 2.43 ähnelt stark der aus dem „FlowPane“, nur das der Zeilenumbruch erzwungen ist. Eine weitere wichtige Eigenschaft von „GrindPane“ ist, dass die Spalten immer in einer Linie untereinander definiert sind. Falls man jedoch die einzelnen Zellen je Zeile dynamisch breit gestalten will, bietet sich eine Befüllung einer „VBox“ mit einer „HBox“ je Zeile an.

2.2.5.4 AnchorPane in Java FX (vgl. [Org14], vgl. [Gor])

In Java FX heißt das Panel, welches ähnliche Funktionen zum „SpringLayout“ aus Java-Swing ermöglicht **AnchorPane**. Dieser durchwegs treffendere Name besagt, dass jede enthaltene Komponente mit einem Anker an einer bestimmten Richtung des Panels hängt. Die Möglichkeit sich mit einem Anker an eine andere im Panel enthaltene Komponente zu hängen, ist nicht mehr vorhanden. Diese fehlende Eigenschaft erzeugt zwar mehr Überprüfungsaufwand bei der Erzeugung – verursacht aber bei der Anzeigekonfiguration keine Einbüßen.

Die Konfiguration dieses Panels sollte im Gegensatz zu allen anderen nicht ausschließlich über den Builder erzeugt werden. Dem **AnchorPaneBuilder** können zwar die enthaltenen Komponenten übergeben werden, eine Positionierung ist allerdings erst am erzeugten Objekt möglich.

```
...
AnchorPane anchorPane = AnchorPaneBuilder.create()
  .children(label1, label2, label3, label4)
  .build();

  // Navigation links in der Anzeige
  anchorPane.setTopAnchor(label1, 15.0);
  anchorPane.setLeftAnchor(label1, 5.0);
  anchorPane.setTopAnchor(label2, 30.0);
  anchorPane.setLeftAnchor(label2, 5.0);

  // Content rechts neben der Anzeige
  // gehoert das gesamte restliche Fenster
  anchorPane.setTopAnchor(label3, 15.0);
  anchorPane.setLeftAnchor(label3, 90.0);
  anchorPane.setRightAnchor(label3, 0.0);
  // verhindert ein moegliches kolidieren mit Label4
  anchorPane.setBottomAnchor(label3, 20.0);
```

```
19
   // Zum Beispiel Button Bar
21 anchorPane.setRightAnchor(label4, 5.0);
   anchorPane.setBottomAnchor(label4, 5.0);
23 ...
```

Quellcode 2.44: erzeuge AnchorPane in Java FX

Wichtig ist es vorauszudenken, wie sich wohl die einzelnen Labels in die Quere kommen könnten, falls sich die Fenstergröße unglücklich verändern würde. Es ist besser, dass einzelne Komponenten keinen Platz mehr haben, bevor Sie sich überlappen – was einen äußerst unprofessionellen Eindruck erweckt. Der Quellcode 2.44 erzeugt in einem Fenster, welches groß genug ist die Anzeige aus Abbildung 2.13.

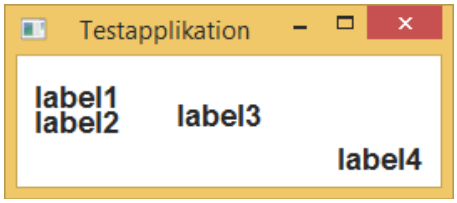


Abbildung 2.13: AnchorPane in Java FX

Falls sich die Anzeige allerdings verkleinern sollte, verschwindet „label3“, bevor es sich mit „label4“ überlappt. Somit können mit diesem Panel einzelne Bereiche in der Hauptanzeige einer Applikation – falls das „BorderPane“ mit seiner Aufteilung nicht ausreicht – sehr schön und leicht unterteilt werden.

3 Systemkonzept

Jetzt da die Technologieanalyse abgeschlossen ist, gilt es zu definieren, wie beim Technologievergleich vorgegangen wird. Im ersten Schritt werden die Vergleichsparameter definiert. Diese Vergleichsparameter berücksichtigen die technische, zeitliche aber auch anwendungsorientierte Ebene. Anschließend wird eine Testapplikation mit verschiedenen Aufgabestellungen definiert, die sowohl mit dem klassischen Entwicklungsweg, als auch mit den modernen, erarbeiteten Technologien umgesetzt wird. Danach wird diese Testapplikation anhand der davor definierten Vergleichsparametern analysiert.

3.1 Vergleichsparameter

3.1.1 Entwicklungsvergleich

Die Testapplikation – die im nächsten Unterkapitel genauer erläutert wird – wird mittels alter und neuer Vorgehensweise entwickelt, und anschließend anhand der hier beschriebenen Vergleichsparameter untersucht.

Die Applikation wird in drei Schritten analysiert, die alle in der praktischen Welt ihre Relevanz haben.

1. **Entwicklung:** Zuerst wird eine Applikation von Grund auf entwickelt. Hier ist es wichtig, dass die Entwicklung sowohl in einer vorgegebenen Zeit abgeschlossen ist, als auch, dass das Ergebnis eine gewisse Modularität, Lesbarkeit und Qualität aufweist.
2. **Anwendung:** Anschließend – nach dem ersten Release – muss nicht nur die Performance passen, sondern auch die Oberfläche ansprechend genug sein (wobei hier stark zwischen interner und externer Software unterschieden wird). Während bei interner Software die Usability im Vordergrund steht, scheint es bei öffentlicher Software oft, dass die grafischen Design-Eigenschaften wichtiger sind und mehr User anlocken. In dieser Arbeit wird ausschließlich der Usability Faktor betrachtet.
3. **Erweiterung:** Es ist eine Illusion, wenn man davon ausgeht, dass eine Klasse einmal erstellt und nie wieder erweitert wird. Jede Software, die genutzt wird, muss irgendwann erweitert oder angepasst werden. Darum hat die Erweiterbarkeit einer Applikation und der damit verbundene Aufwand auf lange Sicht eine sehr große Bedeutung.

In den folgenden Unterkapiteln werden für jeden einzelnen dieser drei Punkte verschiedene sinnvolle Vergleichskriterien definiert und beschrieben.

3.1.1.1 Vergleichsparameter bei Entwicklung

Hier wird wie bereits beschrieben auf die Entwicklungsphase eines neuen Projekts eingegangen.

Quantität der Entwicklung

Der Aufwand bei der Erstellung einer Software muss in einem sinnvollen Zeitrahmen stattfinden. Dieser Zeitrahmen setzt sich meistens aus folgenden beiden Punkten zusammen:

- Einarbeitungszeit
- Entwicklungszeit

Die Aufteilung unter diesen beiden Punkten ist völlig egal, solange die gesamte Entwicklungszeit die Vorgabe nicht überschreitet. Bei kleinen Projekten – für die wenig Zeit anberaumt wird – macht es oft aus Kostengründen keinen Sinn neue Technologien zu analysieren. Bei größeren Projekten hingegen kann durch die Anwendung neuer Technologien bei der Entwicklungszeit dermaßen eingespart werden, dass sich die Einarbeitungszeit schon während des ersten Projektes amortisiert. Aufgrund der Tatsache, dass es sich bei der Testapplikation um ein kleines Programm handelt, werden diese beiden Zeitaufwände unabhängig von einander betrachtet.

Qualität des Quellcodes

Meistens sind neue Technologien durch das Einarbeiten zwar mit einem höheren Zeitaufwand verbunden, dafür muss weniger durch den User selbst entwickelt werden. Dadurch bleibt der Quellcode schlank und leserlich. Diese Eigenschaft ist ein wichtiges Qualitätsmerkmal, da der Code nicht nur übersichtlich – sondern auch für andere Entwickler leserlich und erweiterbar bleibt. Es soll analysiert werden, ob der Vorteil hier wirklich groß ist, oder ob bei sinnvollem, modularen Entwickeln mit dem klassischen Weg auch hier ein vergleichsweise übersichtlicher Code erzeugt werden kann. Dafür wird mit dem Analysetool SonarQuebe[21] anhand folgender Parameter analysiert (vgl. [son13]):

- Klassenanzahl
- Methodenanzahl
- Zeilenanzahl

[21] gängiges Analysetool für Quellcode, was in vielen Projekten für die Qualitätssicherung und für die Sicherung eines standardkonformen Entwicklungsstils eingesetzt wird

- Komplexitätsgrad

Abgesehen davon wird eine subjektive Wertung der Codequalität abgegeben.

Analysequalität

Wichtig ist es bei Fehlern in der Applikation schnell reagieren zu können. Fehler auf die schnell reagiert werden muss sind oft solche, die bei der Entwicklung übersehen wurden und erst in der Produktion auftreten. Hier spielt der Logging-Output für die Fehleranalyse beim Troubleshooting die wichtigste Rolle.

Anschließend kann man in der JRE[22] die Applikation debuggen und versuchen den Fehler somit nachzubilden und anschließend zu beheben. Demzufolge werden hier die folgenden beiden Punkte betrachtet:

- Logging
- Debugging

3.1.2 Laufzeitvergleich

Sobald eine Software fertig entwickelt ist, und an den Kunden ausgeliefert wurde, muss Sie auch noch dessen Anforderungen erfüllen die über eine punktuelle Analyse des Codes hinausgehen.

3.1.2.1 Vergleichsparameter der Performance

Oft wird modernen Technologien – die dem Entwickler viel Quellcode ersparen – vorgeworfen, dass diese durch ihre einfache Bedienbarkeit massiven Overhead erzeugen, der zu gravierendem Performanceverlust führt.

Rechenzeit

Da es sich hier um Technologien und Entwicklung einer Client-Anwendung handelt, die nicht echtzeitorientiert arbeitet, spielt die Performance nur eine untergeordnete Rolle. Zwar darf die Performance nur selten unter eine gewisse Qualitätsuntergrenze fallen – die meist der Reaktionszeit des Benutzers entspricht – allerdings spielt es keine große Rolle ob die Transaktion in 0,5 oder 0,8 Sekunden abgeschlossen wird.

[22] Java Runtime Environment - die Entwicklungsumgebung

Ein Endbenutzer hat auch durchwegs Verständnis, wenn er für mehrere tausend Datenzeilen eine Hand voll Sekunden warten muss – in den meisten Fällen findet sich die Rechenzeit aber sowieso weit unter der Reaktionszeit des Endbenutzers.

Bei diesem Punkt geht es darum zu analysieren, ob sich die Performance bei Massenanforderungen (mehrere tausend Datenzeilen in einer komplexen Tabelle aufbereitet) stark unterscheidet, oder ob die Performance sich relativ ähnlich verhält.

Speicherverbrauch im Betrieb

Neben der Rechenzeit spielt der Verbrauch des Arbeitsspeichers möglicherweise sehr wohl eine große Rolle. Heutzutage ist es in größeren Unternehmen oft der Fall, dass auf dem Arbeitsplatz der Angestellten nur mehr ein Thin-Client steht. Ein Terminal, welches nur ein Mini-Betriebssystem enthält, um sich zum Citrix[23]-Server zu verbinden, auf welchem sich das eigentlich verwendete Betriebssystem befindet. Hier wird der Server-Arbeitsspeicher dynamisch – je nach Bedarf – zwischen den virtuellen Maschinen verteilt. Falls allerdings ein Citrix-Server mehrere hundert Systeme verwaltet, die alle dieselbe Java-Anwendung verwenden, kann hier ein massiver Arbeitsspeicherverbrauch zu einem Problem werden.

Hier werden anhand des Tools Java-Visual-VM folgende Punkte analysiert:

- Anzahl der erzeugten Objekte
- Speicherverbrauch in den einzelnen Phasen der Applikation

3.1.2.2 Vergleichsparameter Release

Hier wird im Gegensatz zum Speicher im Betrieb, der verbrauchte Speicher auf der Festplatte betrachtet. Abgesehen der verbrauchten Speichermenge wird auch verglichen, wie viele Files insgesamt notwendig sind, um die Applikation zum Laufen zu bringen. Bei einer selbstentwickelten Applikation die nur auf Standardressourcen zugreift, müsste eigentlich ein einziges Jar-File ausreichen.

3.1.2.3 Vergleichsparameter des UI Interface

Hier geht es nicht darum, dass die Oberfläche optisch ansprechend ist. Diese Eigenschaft lässt sich nicht sinnvoll analysieren, da Geschmäcker verschieden sind. Was sich allerdings leicht ermitteln lässt, ist ob eine Anzeige standardmäßig zusätzliche Features ermöglicht, die nicht extra entwickelt werden mussten – wie zum Beispiel die Sortierung

[23] ein Virtualisierungssystem von VMWare welches große Verbreitung und Beliebtheit genießt

bei einer Tabelle. Es ist auch wichtig zu analysieren, ob sich die einzelnen Komponenten richtig aktualisieren, ob die Interaktion mit dem Benutzer richtig funktioniert und ob die Anzeige sich bei verschiedenen Fenstergrößen immer korrekt verhält.

3.1.3 Erweiterbarkeitsvergleich

Nachdem die Applikation fertiggestellt wurde, kommt es eigentlich immer zu Änderungswünschen oder Erweiterungen. Dieser Punkt ist der Wichtigste von allen. Die Erweiterbarkeit – und der damit verbundene Aufwand – spielen eine tragende Rolle bei der Technologieauswahl.

Quantität der Entwicklung

Es macht keinen Sinn eine Anwendung schnell zu entwickeln, bei der jede Erweiterung extrem viel Zeit in Anspruch nimmt und Veränderungen in großem Ausmaß bedingt. Darum wird hier gegenübergestellt, wie lange die selbe Erweiterung bei welcher Entwicklungsmethode in Anspruch nimmt.

Qualität des Quellcodes

Abgesehen von dem Zeitaufwand muss auch gesagt sein: Je mehr Zeilen hinzugefügt werden, oder abgeändert werden, desto höher ist das Risiko einen neuen Fehler einzubauen und desto genauer muss die neue Implementierung getestet werden. Darum ist es wichtig, dass eine Applikation so einfach wie möglich und mit so wenigen Befehlsfolgen wie möglich erweitert werden kann.

3.2 Anwendungsspezifikation

Es soll eine Testanwendung entwickelt werden, welche den Vergleich in folgenden Punkten ermöglicht:

- Layouting
- Interaktion der Panels
- komplexe Tabellen
- Diagramme
- Filter

Darum entsteht eine Applikation, die eine Tabelle mit mehreren tausend Transaktionen enthält. Die Tabelle mit den Transaktionen beschreibt den Kern der Anzeige. Unter die-

ser Tabelle befindet sich ein Diagramm, welches eine Gegenüberstellung der Einnahmen und Ausgaben je Jahr anzeigt. Die Anzeige zeigt links oben auch die Summierung aller Transaktionen je Konto – also den aktuellen Kontostand – an. Abbildung 3.1 zeigt die Vorlage für die Anordnung der einzelnen Programmteile.

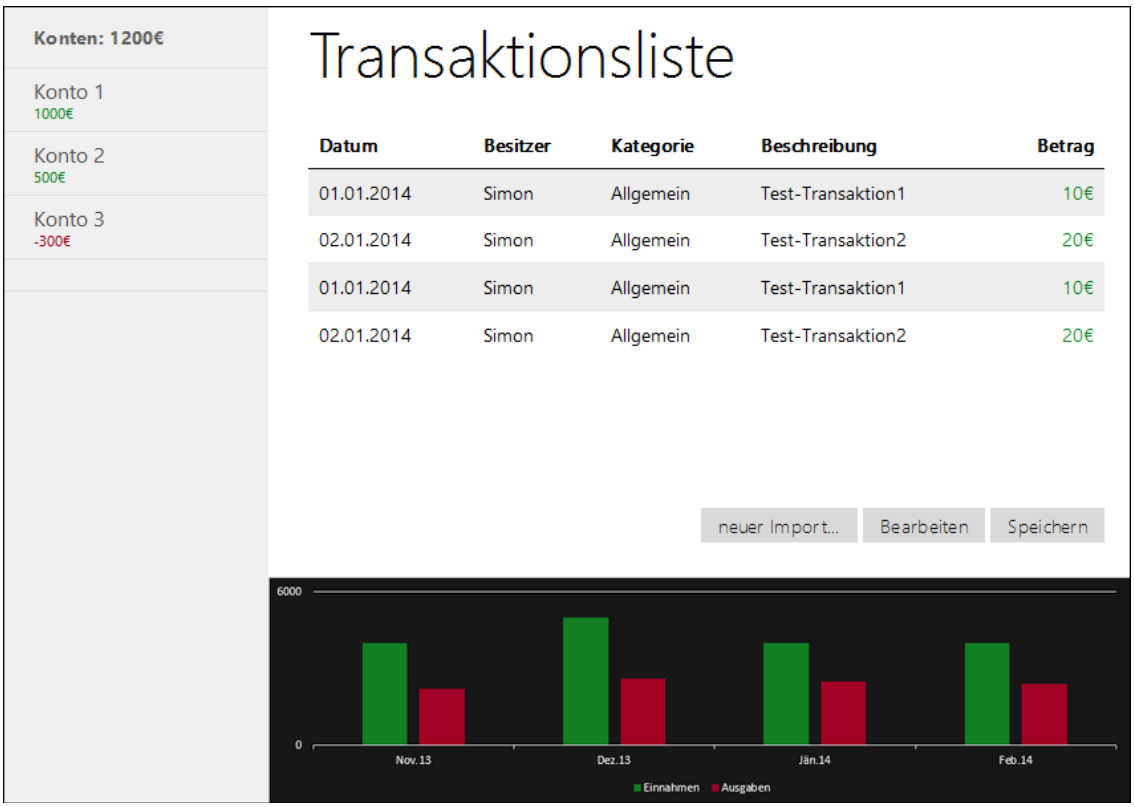


Abbildung 3.1: zu erzeugende Applikation

3.2.1 Beschreibung der Datenbank

Die Datenbank ist von der Entwicklung der Applikationen losgelöst. Beide Entwicklungsvarianten – klassisch als auch modern – verbinden sich mit einer MySQL-Datenbank. Als Datenbankmodell wird die Datenbank aus dem Praxisprojekt verwendet. Abbildung 3.2 zeigt das ER-Modell. Diese Datenbank ist mit mehreren tausend Testdaten befüllt um einen Sinnvollen Vergleich zu ermöglichen. (vgl. [War13b], S. 9 - 14)

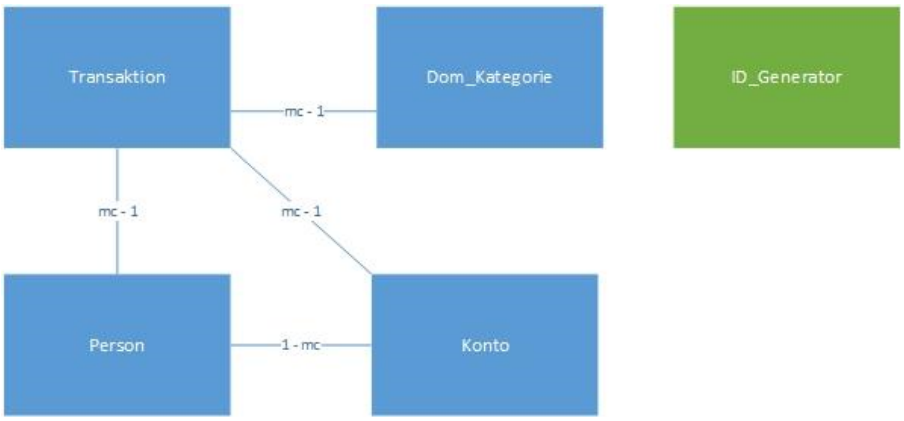


Abbildung 3.2: Datenbankmodell

Tabelle Dom_Kategorie

Diese Tabelle beschreibt in welche Kategorie eine Transaktion gehört. Es gibt zum Beispiel: Allgemein, Essen, Haushalt, Auto usw. Diese Tabelle besteht aus nur einer Spalte „DOMVALUE“ auf die von Seiten Tabelle „TRANSAKTION“ referenziert wird. Somit schränkt diese Tabelle lediglich die Werte der dazugehörigen Spalte in der Tabelle „TRANSAKTION“ ein.

Tabelle Person

Eine Person ist sowohl Besitzer einer Transaktion, als auch eines Kontos. Es ist nicht zwingend notwendig, dass die Transaktion dem Kontobesitzer gehört. Es könnte zum Beispiel vorkommen, dass ein Kind mit der Kontokarte der Eltern bezahlt. Diese Konstellation ist durchwegs dynamisch. Die Tabelle enthält die Spalten ID, VORNAME, NACHNAME und USERNAME (Anzeigename in der Oberfläche).

Tabelle Konto

Ein Konto gehört genau einer Person. Jede Transaktion wird auf einem Konto durchgeführt. Die Tabelle enthält die Spalten ID, IBAN, FPERSON (Kontoinhaber) und NAME (Anzeigename des Kontos).

Transaktion

Diese Tabelle stellt den Kern der Testapplikation dar. Der Inhalt dieser Tabelle wird auch als Mittelpunkt der Anzeige gewählt – wobei nur ausgewählte Spalten angezeigt werden.

In dieser Tabelle bedarf es bei jeder Spalte einer kurzen Erklärung, wieso diese für diese Testapplikation von Relevanz ist.

Spaltenname	Beschreibung
ID	Numerischer Primärschlüssel, der durch Java vergeben wird. Wert wird aus der Tabelle ID_GENERATOR geholt.
FPERSON	beschreibt den Besitzer der Transaktion
FDOM_KATEGORIE	beschreibt die Kategorie zur Transaktion
TEXT	Beschreibt den Banktext, der in der Oberfläche nicht angezeigt wird.
NOTIZ	Enthält die vom Benutzer hinzugefügte Beschreibung der Transaktion.
DURCHFUEHRUNGS_DATE	Das Datum an dem die Transaktion durchgeführt wurde.
BETRAG	der Betrag der Transaktion als Gleitkommazahl mit zwei Nachkommastellen
FKONTO	das Konto zu der Transaktion; ein Foreign-Key der in der Anzeige nicht aufscheint

Tabelle 3.1: Tabelle Transaktion und deren Spalten

3.2.2 Anforderung Panel Transaktionsliste

Im Zentrum der Anzeige befindet sich eine Tabelle aller Transaktionen. Das Panel enthält die statische Überschrift, die Tabelle in einem Scroll-Panel und die darunter befindlichen Buttons.

Tabelle Transaktionsliste

Es soll für diese Tabelle eine Framework-Komponente erzeugt werden, die das einfache Erzeugen von komplexen Tabellen ermöglicht. Diese Tabelle soll das Anzeigen, Editieren und Speichern des Inhalts ermöglichen. Ein solcher Schritt ist für diese Testapplikation, in der nur eine Tabelle vorkommt eigentlich nicht notwendig. Allerdings werden in jedem größeren Projekt eigens entwickelte Framework-Komponenten verwendet. Es soll auf diese Art verglichen, werden ob in Java FX leichter universell einsetzbare, intelligente Tabellen erzeugt werden können.

Dieses Panel soll unter Verwendung dieser intelligenten Tabelle entstehen. Die Tabelle zeigt ausgewählte Spalten, aber alle Zeilen der Datenbanktabelle „TRANSAKTION“ an. Für die einzelnen Spalten gilt die Beschreibung aus Tabelle 3.2.

Anzeigename	Datenbankspalte	Beschreibung
Datum	DURCHFUEHRUNGS_DATE	In der Datenbank ein VARCHAR, in Java ein Datumsobjekt. Ist beim Drücken auf „Bearbeiten“ als Text im Format „dd.MM.yyyy“ eingebbar.
Besitzer	FPERSON	In der Oberfläche wird der Name der Person angezeigt, obwohl in der Spalte selbst deren ID als Foreign-Key steht. Beim neuen Erstellen einer Transaktion muss hier ein Auswahlfeld mit allen vorhandenen Usernamen aufscheinen.
Kategorie	FDOM_KATEGORIE	Hier steht genau der Inhalt der Datenbankspalte geschrieben. Beim Erzeugen einer neuen Transaktion muss das Auswahlfeld allerdings mit allen möglichen Domänenwerten befüllt werden.
Beschreibung	NOTIZ	Hier steht – soweit es die Breite der Anzeige zulässt – der durch den User erfasste Text der Transaktion. Beim Bearbeiten steht die Eingabe als Textfeld.
Betrag	BETRAG	Hier steht der Betrag. Positive Zahlen sind grün (inklusive Null), negative rot. Beim Bearbeiten steht auch hier ein Textfeld.

Tabelle 3.2: Beschreibung der Spalten der Transaktionsanzeige

Buttons zur Tabelle

Unter der Tabelle befinden sich drei Buttons – wobei der Button „Speichern“ standardmäßig ausgegraut ist. Die gesamte Tabelle ist beim Starten der Applikation nicht zur Bearbeitung aktiviert. Erst beim Drücken auf „Bearbeiten“ lässt sich jeder angezeigte Wert manipulieren. Beim Drücken auf „Speichern“ werden die eingegebenen Werte in die Datenbank übernommen und der Bearbeiten-Modus wird wieder beendet. Bis zur Beendigung des Bearbeiten-Modus ist der Button „Bearbeiten“ inaktiv und der Button „Speichern“ aktiv.

Beim Drücken auf „neuer Import...“ müssen alle in der Oberfläche nicht angezeigten Datenbanktabellenspalten über Dialoge abgefragt werden. Anschließend wird eine leere Zeile in die Oberfläche hinzugefügt, in die die noch offenen Spalten eingetragen werden müssen. Dazu wird automatisch in den Bearbeiten-Modus gewechselt.

3.2.3 Anforderung Diagramm-Panel

Unter dem Tabellen-Panel befindet sich ein Diagramm, welches unabhängig vom Filter über alle Transaktionen hinweg eine Gegenüberstellung von Einnahmen und Ausgaben pro Monat erstellt.

Dieses Diagramm soll – falls möglich – nach rechts scrollen können, damit die Anzeige nicht zu eng zusammengedrückt wird. Eine Anzeige über zum Beispiel drei Jahre – also 36 Monate – auf einmal würde die Erlebnisse nicht sinnvoll darstellen.

3.2.4 Anforderung Summen-Panel

Zu guter Letzt gibt es ein Summen-Panel links oben im Fenster. Dieses Panel zeigt zuerst die Summe über alle Konten und anschließend den individuellen Kontostand an. Sobald eine neue Transaktion hinzugefügt wird, oder eine Transaktion bearbeitet wurde, muss diese Anzeige ihre Werte aktualisieren.

3.3 Vorgehensweise der Entwicklung

Im nächsten Kapitel wird zuerst die Datenbankabstraktion und anschließend iterativ der Aufbau jedes der beschriebenen Panels entwickelt – zuerst auf klassischem Weg und anschließend unter Verwendung der analysierten Technologien. Es wird jeweils auf spezielle Eigenschaften beim Entwickeln eingegangen – der gesamte Quellcode befindet sich auf einer CD im Anhang.

Anschließend wird in Kapitel 5 zu jedem dieser Punkte einzeln und anschließend zur gesamten Applikation die Analyse nach der Beschreibung aus Kapitel 3.1 erstellt und bewertet. Somit können Schwächen und Stärken in den einzelnen Gebieten – als auch eine endgültige globale Entscheidung präsentiert werden.

4 Entwicklung

In diesem Kapitel wird auf die speziellen Eigenschaften der Entwicklung der Test-Applikationen eingegangen. Zuerst wird auf nötige Basisklassen eingegangen, die für beide Projekte notwendig sind. Anschließend wird die Datenbankanbindung und Abstraktion thematisiert. Danach wird der Aufbau der einzelnen Panels aus Kapitel 3.2 beschrieben und deren Zusammenführung thematisiert. Im letzten Schritt wird auf die Erweiterbarkeit der Applikationen eingegangen.

Die hier erarbeiteten Applikationen dienen als Grundlage für den Vergleich im nächsten Kapitel. Der gesamte Quellcode befindet sich auf einer CD im Anhang.

Was die Namenskonvention betrifft, werden gemeinsam genutzte Klassen mit „DS“[24] begonnen. Klassen aus dem Projekt der klassischen Entwicklung beginnen mit „DO“[25]. Hingegen dazu beginnen die Klassen der mit modernen Technologien entwickelten Applikation mit „DM“[26].

4.1 Vorbereitung und Service-Klassen

In diesem Kapitel werden einzelne Dienste innerhalb der Java-Projekte beschrieben, auf die in der gesamten Applikation zugegriffen wird.

Diese Dienste werden sowohl von der Datenbankabstraktionsebene, als auch der grafischen Oberfläche genutzt – spezifische Service-Klassen werden in den eigens dafür erstellten Kapiteln erläutert.

Hier geht es vor allem um folgende Punkte:

- Logging
- Exceptions
- Event-Handling

[24] Diplomarbeit – Shared
[25] Diplomarbeit – Old
[26] Diplomarbeit – Modern

4.1.1 Logging in den Applikationen

4.1.1.1 DO-Entwicklung

Hier muss das gesamte Logging-Konstrukt nachgebildet werden. Dafür wurde die Klasse **DOLogger** geschrieben, mit der via Übergabeparameter ein Log-Level übergeben werden kann.

Es werden die Methoden „info()“, „warn()“, „error()“ und „debug“ zur Verfügung gestellt, deren Meldungen je übergebenem Log-Level zwischen 0 (nur Error) und 3 (alles bis Debug) ausgegeben werden. Außerdem ist es möglich die Log-Meldungen in ein File zu pipen – mittels der Methode „setLogFile()“. Ein Logger-Aufruf sieht wie in Quellcode 4.1 beschrieben aus.

```
1       // setze Startlevel
        DOLogger.setLevel(DOLogger.LEVEL_DEBUG);
3       // gebe Meldung, ausgehend von Klasse DOLogger aus
        DOLogger.info("Testmeldung", DOLogger.class);
```

Quellcode 4.1: DO Loggeraufruf

4.1.1.2 DM-Entwicklung

Hier wird die Technologie „Log4J“ verwendet. Um das Loggen zu aktivieren, muss lediglich ein Konfigurations-File **log4j.properties** erzeugt und definiert werden.

Im Vergleich zum selbst entwickelten Logger wird hier die ausgehende Klasse automatisch ermittelt. Auch ein Setzen des Log-Levels muss nicht über einen Methodenruf erfolgen, da das Property-File beim Start der Applikation automatisch eingelesen wird. Abgesehen davon gibt es bei Log4J die Möglichkeit verschiedene Logger zu verwenden, und diesen unabhängig von einander Log-Level zu geben. In dieser Arbeit wird nur ein Logger verwendet.

```
    Logger.getLogger("LOGGERNAME").info("Testmeldung");
```

Quellcode 4.2: DM Loggeraufruf

4.1.2 Exceptions in den Applikationen

4.1.2.1 DS-Entwicklung

Bei beiden Applikationen – sowohl klassisch als auch modern – werden die selben Exceptions verwendet.

Klassenname	Beschreibung
DSUserException	Diese Exception soll geworfen werden, falls der Applikationsbenutzer sich falsch verhält und es dadurch zu einem Fehler kommt. Zum Beispiel bei der Eingabe eines Textes in ein Zahlenfeld.
DSAppException	Beschreibt Fehler, die durch die Applikation selbst auftreten und somit nicht vorhersehbar waren.

Tabelle 4.1: Exceptions in Testapplikation

4.1.3 Eventhandling in den Applikationen

4.1.3.1 DS-Entwicklung

Auch hier gibt es eine gemeinsam verwendete Ressource. Es gibt verschiedene Event-Arten auf, die ein Objekt aufmerksam gemacht werden muss. Darum wurden hierfür folgende Listener in Tabelle 4.2 – die jeweils auf ein Event warten – definiert.

Klassenname	Beschreibung
DSRefreshEventListener	Listener horcht, ob es in der Datenbank Änderungen gab, von denen das Objekt abhängig sein könnte (zum Beispiel um die Anzeige aktueller Zahlen garantieren zu können). Es wird hierfür im „DSRefreshEvent“ eine Liste der betroffenen Datenbanktabellen übergeben, damit die horchende Klasse abschätzen kann, ob Sie von der Änderung betroffen ist.
DSStatusEventListener	Listener horcht auf den Status der gesamten Applikation. Dieser kann entweder „OK“ oder „IN_BEARBEITUNG“ sein. Eine Applikation, die sich zum Beispiel gerade in Bearbeitung befindet darf nicht geschlossen werden. Der Benutzer muss erst den Bearbeitungszustand abschließen.

Tabelle 4.2: Listener in Testapplikation

Wenn ein Objekt einen Listener benötigt, muss es diesen an der Klasse **DSEventBus** anhängen. Hier können auch Events abgefeuert werden, falls ein Objekt Daten verändert hat und diese Information über die gesamte Applikation kommunizieren will. Sämtliche Methoden der Bus-Klasse sind statisch und können daher ohne Instanziierung aufgerufen werden.

4.2 Entwicklung der Datenbankabstraktion

Hier wird für die modern entwickelte Anwendung die in Kapitel 2.1 erarbeitete Technologie JPA verwendet. Für die klassische Applikation werden in den meisten Fällen direkte JDBC-Statements verwendet, die gegebenenfalls Java-Datenobjekte befüllen. Hierfür wird die im Praxisprojekt (siehe [War13b]) erarbeitete Vorgehensweise als Vorlage verwendet.

4.2.1 Datenbankanbindung in den Applikationen

4.2.1.1 DO-Entwicklung

Die gesamte Datenbankkommunikation wird – der Übersichtlichkeit halber – über eine Klasse zentral gesteuert, die **DODatabaseConnection**. Die Klasse ist nach dem Pattern „Singleton“[27] aufgebaut. Beim ersten Zugriff wird die Datenbankverbindung konfiguriert und aufgebaut. Es stehen drei öffentliche Methoden zur Verfügung die in Tabelle 4.3 beschrieben sind.

Methodenname	Beschreibung
getInstance()	gibt die Singleton-Instanz der Klasse zurück
createStatement()	erzeugt ein Statement und gibt dieses zurück
createPreparedStatement(String sql)	erzeugt ein vorbereitetes Statement und gibt dieses zurück
getConnection()	Falls mehr gemacht werden muss als nur Statements auszuführen, kann man sich auch die Connection direkt zurückgeben lassen.

Tabelle 4.3: Datenbankanbindung in DO-Applikation

4.2.1.2 DM-Entwicklung

Auch hier wird die Verbindung zur Datenbank an einer Stelle verwaltet – aus der selben Intention heraus wie bei der DO-Applikation. Die Klasse **DMDatabaseConnection** basiert auf den Erkenntnissen aus einer anderen eigens entwickelten Arbeit. (vgl. [War13a], S. 21f)

Deren einzige öffentliche Methode **getEntityManager()** gibt den „EntityManager“ zurück. Somit ist die Aufgabe dieser Singleton-Klasse beim ersten Zugriff das Konfigurationsfile „persistence.xml“ einzulesen (siehe Kapitel 2.1.4.1) und den fertigen „EntityManager“ zurückzugeben.

[27] es gibt von Klassen nach Singleton nur eine Instanz die beim Erstzugriff initialisiert wird

4.2.2 Primary-Key Verwaltung in den Applikationen

4.2.2.1 DO-Entwicklung

Hier wurde die Klasse **DOIdGenerator** entwickelt die eine „Sequence“ nachbildet. Ähnlich wie in JPA wird der aktuelle Wert einer Zahl aus einer Schlüsseltabelle gelesen und anschließend erhöht. Der ausgelesene Wert kann durch den Entwickler als Primärschlüssel verwendet werden.

Die Klasse bietet die statische, öffentliche Methode **getNextVal(String table)** an, der man den Namen der Tabelle übergibt für welche der ermittelte Wert bestimmt ist. Zurückgegeben wird besagter Integer-Wert, der als Primärschlüssel verwendet werden kann.

4.2.2.2 DM-Entwicklung

In JPA wird die Primärschlüsselvergabe via Annotations geregelt. Eine eigene Implementierung ist nicht notwendig. Siehe hierfür das Kapitel 2.1.3.3.

4.2.3 Datenbankzugriff in den Applikationen

4.2.3.1 DO-Entwicklung

Hier muss der gesamte Zugriff manuell erzeugt werden. Aus Gründen der Übersichtlichkeit und Modularität gibt es je Datenbanktabelle ein einfaches Datenobjekt welches nur aus Properties besteht. Daraus folgen die vier POJO’s

- DOPerson
- DOKonto
- DODomKategorie
- DOTransaktion

Diese POJOs werden bei der Tabellendarstellung auf jeden Fall in der Oberfläche benötigt. Falls keine Datenobjekte benötigt werden, kann man im Quellcode einfach über die Home-Klassen der einzelnen Tabellen SQL-Abfragen durchführen. In diesen Home-Klassen wird für jede gängige SQL-Abfrage eine eigene Methode geschrieben. Dadurch befinden sich die Abfragen nicht quer über den Code verteilt.

Daraus folgt, dass man je Tabelle eine Home-Klasse hat, in der alle Abfragen gesammelt verwaltet werden können. Diese Klassen sind alle nach dem Singleton-Pattern aufgebaut.

- DOPersonHome
- DOKontoHome
- DODomKategorieHome
- DOTransaktionHome

In diesen Homes wird jeweils eine Methode angeboten die einen Datensatz aufgrund einer ID sucht und als POJO retouniert, und eine Methode die alle Datensätze der Tabelle als POJO zurückgibt. Außerdem werden alle gefundenen Datensätze in einer Map gebuffert, um keinen mehrfachen Zugriff auf die Datenbank bei Abfrage des selben Datensatzes zu erzeugen. Allerdings sind somit diese gebufferten Werte nicht mehr aktuell – da hier keine Synchronität zur Datenbank besteht. Diese Map kann mit dem Befehl „clean()“ gelöscht werden, was auch nach jeder Abfrage getan werden sollte. Quellcode 4.3 zeigt ein exemplarisches Beispiel wie viel Entwicklungsaufwand je Datenbanktabelle nur für diese drei Methoden entsteht.

```
1  ...
   /**
3   * gebufferte Konten - kann mittels CLEAN() entleert werden
    */
5  private HashMap<Integer, DOKonto> kontoMap = new HashMap<Integer, DOKonto>();

7  /**
    * entleert cache
9   */
   public void clean(){
11   kontoMap.clear();
   }
13
   /**
15  * gibt ein einzelnes Konto zurueck
    * @param id - Primaerschluessel des Kontos
17  * @return Konto als POJO
    * @throws SQLException - falls bei der Suche ein Fehler auftritt
19  */
   public DOKonto findKontoById(Integer id) throws SQLException{
21   return findKontos(" where id = " + id).get(0);
   }
23
   /**
25  * gibt alle Konten zurueck
    * @param wherePart - SQL-Query ab "where" - zB. where 1 = 1
27  * @return ArrayList mit Datensaetzen als POJOs
    * @throws SQLException - falls bei der Suche ein Fehler auftritt
29  */
   public ArrayList<DOKonto> findKontos(String wherePart) throws SQLException{
31   ArrayList<DOKonto> retVal = new ArrayList<DOKonto>();
     Statement st = DODatabaseConnection.getInstance().createStatement();
33   st.execute("select * from konto " + wherePart);
     ResultSet rs = st.getResultSet();
35
     while(rs.next()){
37     Integer id = rs.getInt(1);
       if(kontoMap.containsKey(id)){
39       retVal.add(kontoMap.get(id));
         continue;
41     }
       DOKonto konto = new DOKonto();
43     konto.setId(id);
       konto.setIban(rs.getString(2));
```

```
45        konto.setFperson(DOPersonHome.getInstance().findPersonById(rs.getInt(3)));
          konto.setName(rs.getString(4));
47        retVal.add(konto);
          kontoMap.put(id, konto);
49      }

51      st.close();
        return retVal;
53    }
...
```

Quellcode 4.3: DO Datensuche

Abgesehen davon bietet jede Home-Klasse die Möglichkeit ein neues POJO mittels „create()“ – welches danach nur eine ID enthält – zu erzeugen. Dieses POJO kann nach Befüllung mittels „store()“ in die Datenbank gespeichert werden. Datensätze die aktualisiert wurden können auch mittels dieser Methode an die Datenbank weitergereicht werden.

```
...
2   /**
     * erzeugt ein POJO mit befuellter ID
4    * @throws SQLException - falls ein Fehler auftritt
     * @return gibt ein neues POJO inkl. ID zurueck
6    */
    public DOKonto create() throws SQLException{
8     DOKonto konto = new DOKonto();
      konto.setId(DOIdGenerator.getNextVal("konto").intValue());
10    return konto;
    }
12
    /**
14   * speichert das uebergebene POJO
     * @param konto - ein aktualisiertes oder neues Konto
16   * @throws SQLException - falls ein Fehler auftritt
     */
18  public void store(DOKonto konto) throws SQLException{
      Statement st = DODatabaseConnection.getInstance().createStatement();
20
      if(findKontos(" where id = " + konto.getId()).size() == 0){
22      String sql = "insert into konto values ( " +
          konto.getId() + ", " +
24        " '" + konto.getIban() + "', " +
          konto.getFperson().getId() + ", " +
26        " '" + konto.getName() + "')";
        st.executeUpdate(sql);
28
      }else{
30      String sql = "update konto set " +
          " iban = '" + konto.getIban() + "', " +
32        " fperson = " + konto.getFperson().getId() + ", " +
          " name = '" + konto.getName() + "' " +
34        " where id = " + konto.getId();
        st.executeUpdate(sql);
36    }

38    st.close();
    }
40  ...
```

Quellcode 4.4: DO Datenerzeugung und Speicherung

Die Tabelle aus diesem Beispiel enthält lediglich vier Spalten, aber bereits hier entstehen bei Quellcode 4.3 und 4.4 viele Entwicklungszeilen. Je größer die Tabelle, umso höher der Aufwand und die Fehleranfälligkeit. Jede dieser Methoden – da manuell erstellt – muss genau auf ihre Funktionstüchtigkeit überprüft werden. Diese Methoden dienen allerdings nur als Vorschlag und müssen nicht verwendet werden. Es kann genauso alles direkt über Statements abgewickelt werden.

4.2.3.2 DM-Entwicklung

Die JPA-POJOs wurden wie im Kapitel 2.1.2 bis 2.1.4 beschrieben erzeugt. Dieser Schritt erfolgte mit sehr geringem Zeitaufwand und ohne Probleme. Der Datenbankzugriff kann somit wie in Quellcode 4.5 beschrieben bei der GUI-Entwicklung mit geringen Aufrufen erfolgen.

```
...
EntityManager em = DMDatabaseConnection.getIntance().getEntityManager();

List<DMPerson> list = em.createQuery(
    " select p from Person p ",
    DMPerson.class).getResultList();
for(DMPerson p:list)
  System.out.println(p.getUsername());
...
```

Quellcode 4.5: DM Datenbankzugriff

Auch die Befüllung, Speicherung und Aktualisierung wird automatisch durch den „EntityManager“ – wie in Kapitel 2.1.6 beschreiben – durchgeführt.

4.3 Entwicklung der Oberfläche

4.3.1 Tabellen Transaktionsliste

4.3.1.1 DO-Entwicklung

Mit Java Swing ist das Entwickeln von Framework-Komponenten immer mit massivem Aufwand verbunden. Um hier eine relativ einfache Erstellung neuer Tabellen zu ermöglichen, wurden die in Tabelle 4.4 beschriebene Klassen erzeugt.

Klassenname	Beschreibung
DOColumn	Beschreibt eine Spalte der Tabelle und wie auf diese zugegriffen wird.
DODataPOJO	Jedes normale Tabellen-POJO, welches der „DOTable“ übergeben wird, muss von dieser Klasse abgeleitet sein.
DODateEditor	Beschreibt den Editor, falls man eine Datumszelle bearbeitet.

DODomEditor	Beschreibt den Editor, falls man eine Domänenzelle bearbeitet.
DODomHandler	Beschreibt hier das Auswahlfeld, welches bei einer Domänenzelle alle Domänenwerte enthält.
DODomPOJO	Domänen oder domänenähnliche Objekte können sich anstatt von „DODataPOJO“ von dieser Klasse ableiten.
DOTable	Beschreibt die fertige Tabelle, welche sich von „JTable“ ableitet.
DOTableModel	Beschreibt das TableModel, welchem die Spalten und der Datenvektor übergeben wird.

Tabelle 4.4: DO-Tabellenklassen

Unter Verwendung der Klassen aus Tabelle 4.4 konnte das Panel **DOTransaktionsTablePanel** erstellt werden. Die Vorgehensweise bei der Erstellung einer Tabelle läuft wie folgt ab.

1. erzeuge Tabellenspaltendefinitionen via „DOColumn“
2. erzeuge „DOTableModel“ und übergebe Spalten und Datenvektor
3. übergebe das Model einer neuen Tabelle vom Typ „DOTable“

Daraus folgt der verhältnismäßig einfacher Quellcode 4.6. Die Buttons müssen manuell entwickelt werden und können bei der Neuerstellung auf die Methode **addNewLine()** und beim Bearbeiten auf **setEditable()** zurückgreifen.

```
...
// ermittle Datenvektor
Vector<DOTranaktion> data = DOTransaktionHome.getInstance().findTransaktionen("");

// definiere Spalten
DOColumn[] cols = new DOColumn[]{
  //Spalte DATUM
  new DOColumn("Datum", "DurchfuehrungsDate", false, .2, Date.class),
  new DOColumn("Besitzer", "Fperson", false, .2, new DODomHandler<DOPerson>() {
    // in diesem Fall - da via Auswahlfeld angezeigt - domaenenaehnlich
    @Override
    public Vector<DOPerson> getDomValues() {
      try {
        return DOPersonHome.getInstance().findPersonen("");
      } catch (SQLException e) {
        DOLogger.error("Fehler beim Suchen aller Personen!", getClass(), e);
      }
      return null;
    }
  }),
  // Spalte KATEGORIE
  new DOColumn("Kategorie", "FdomKategorie", false, .2, new DODomHandler<
      DODomKategorie>() {
...
  }),
  // Spalte BESCHREIBUNG
```

```
  new DOColumn("Beschreibung", "Notiz", false, .4, String.class),
  // Spalte BETRAG
  new DOColumn("Betrag", "Betrag", false, .2, Double.class)
};

// erzeuge Tabellenmodell
tableModel = new DOTableModel<DOTranaktion>(data, cols) {
  // beschreibt Vorgang beim Hinzufuegen einer neuen Zeile
  // -> nicht angezeigte Spalten muessen ueber eine Routine ermittelt werden
  @Override
  public DOTranaktion getNewLine() {
    try {
      DOTranaktion trans = DOTransaktionHome.getInstance().create();
...
      return trans;
    } catch (SQLException e) {
      DOLogger.error("Fehler beim Erzeugen einer neuen Transaktion!", getClass(), e
          );
    }
    return null;
  }
};

// erzeuge Tabelle
table = new DOTable(tableModel);
...
```

Quellcode 4.6: DO Tabellenimplementierung

4.3.1.2 DM-Entwicklung

Hier kann man auf die Entwicklung einer eigenen Framework-Technologie verzichten, da der „TableViewBuilder“ hierfür bereits genug Konfigurationsmöglichkeiten zur Verfügung stellt (siehe Kapitel 2.2.3). Als Datenobjekte für diese Tabelle können direkt die JPA-POJOs verwendet werden, wie Sie in Kapitel 2.1.2 beschrieben sind. Es muss lediglich für jede anzuzeigende Spalte ein Java FX Property wie in Kapitel 2.2.3.2 - 2.2.3.6 hinzugefügt werden. Der Quellcode 4.10 zeigt, wie zum Beispiel die Spalte „notiz“ im JPA-POJO um ein Java-FX Property erweitert werden kann.

```
...
  /**
   * JPA:
   * beschreibt Spalte NOTIZ in Datebank
   */
  @Column(name="notiz", nullable=false, length=400)
  private String notiz;

  /**
   * Java FX:
   * beschreibt Property fuer Spalte NOTIZ fuer TableView
   */
  @Transient
  private StringProperty notizProperty;

  /**
   * erzeugt ggf. neues Property mit den Namen "notiz"
   * @return gibt ein Java FX String-Property zurueck
   */
  public StringProperty notizProperty(){
```

```
    if(notizProperty == null && getNotiz() != null)
      notizProperty = new SimpleStringProperty(this, "notiz", getNotiz());
    else if(notizProperty == null)
      notizProperty = new SimpleStringProperty(this, "notiz");
    return notizProperty;
  }

  /**
   * @return gibt NOTIZ zurueck
   */
  public String getNotiz() {
    return notiz;
  }

  /**
   * setzt NOTIZ
   * @param notiz - zu setzender Wert
   */
  public void setNotiz(String notiz) {
    notizProperty().set(this.notiz = notiz);
  }
...
```

Quellcode 4.7: DM JPA-POJO mit Java FX-Property

Nachdem die POJOs alle erweitert wurden, konnte das Panel **DMTransaktionsTablePanel** ohne Probleme erstellt werden. Das Ansprechen der Buttons (siehe Kapitel 2.2.2.5) funktionierte auch problemlos. Das Hinzufügen eines neuen Elementes konnte durch hinzufügen in die „ObservableList“ ohne größeren Aufwand durchgeführt werden (Kapitel 2.2.2.4). Um zu speichern musste man am „EntityManager“ lediglich ein „commit()“ durchführen, da es sich bei allen Datensätzen der Tabellen um „Managed-Entities“ (Kapitel 2.1.6.2) handelte. Das Layouting mittels „AnchorPane“ (Kapitel 2.2.5.4) funktionierte auch einwandfrei.

4.3.2 Diagramm-Panel

4.3.2.1 DO-Entwicklung

Hier muss alles mittels dem Object „Graphics“ und der Methode „paint()“ selbst gezeichnet werden. Daraus entsteht zwar ein massiver Aufwand allerdings ist das Resultat komplett anpassbar.

Es wurde ein Diagramm **DOBarChart** entwickelt welches – falls breiter als das Fenster – sich mittels Maus nach links und rechts verschieben lässt. Diesem Diagramm muss ein Vector übergeben werden, welcher Klassen des POJOs **DOBarChartObject** enthält. Dieses POJO enthält folgende drei Spalten.

- ein Label unter welchem die Daten angezeigt werden
- einen Einnahme-Wert
- einen Ausgabe-Wert

Da das Diagramm in einem „JPanel“ liegt, kann es ohne Probleme in der Oberfläche eingebettet werden.

```
...
  Vector<DOBarChartObject> v = new Vector<DOBarChartObject>(){{
...
  }};

  DOBarChart chart = new DOBarChart();
  chart.refreshMap(v);
...
```

Quellcode 4.8: DO Diagrammimplementierung

Die Verwaltung und Befüllung dieses Diagrammes erfolgt in der Klasse **DODiagramm-Panel**. Die notwendigen Daten werden direkt über ein SQL-Statement ermittelt und anschließend in die POJOs gefüllt.

```
...
Vector<DOBarChartObject> v = new Vector<DOBarChartObject>();
Statement st = DODatabaseConnection.getInstance().createStatement();
st.execute(" select substr(durchfuehrungs_date, 1, 7) as monat, " +
  " sum(case when betrag > 0 then betrag else 0 end) as betrag_plus, " +
  " sum(case when betrag < 0 then betrag else 0 end) as betrag_minus " +
  " from transaktion " +
  " group by substr(durchfuehrungs_date, 1, 7) " +
  " order by substr(durchfuehrungs_date, 1, 7) asc");

ResultSet rs = st.getResultSet();
while(rs.next()){
  v.add(new DOBarChartObject(
    rs.getString(1).replace("-", ""),
    (int)rs.getDouble(2),
    -(int)rs.getDouble(3)));
}
...
```

Quellcode 4.9: DO Query für Diagramm

Abschließend hängt sich die Klasse noch an den „DSEventBuss“, damit das Diagramm bei möglichen neuen Transaktionen automatisch aktualisiert wird.

4.3.2.2 DM-Entwicklung

Auch hier wird in Java FX – durch genug Standardkonfigurationsmöglichkeiten – keine eigens entwickelte Framework-Komponente benötigt. Es wird direkt in der Klasse **DMDiagrammPanel** ein „BarChart“ Objekt instanziert.

Hier wird – genau wie bei der Summenanzeige (siehe nächstes Kapitel 4.3.3.2) – ein eigenes Datenobjekt **DMDiagrammValue** verwendet. Bei der JPQL-Abfrage in Quellcode 4.10 wird direkt eine Liste mit diesem Objekt befüllt – bestehend aus formatierten Monatsnamen, Einnahmen-Werten und Ausgaben-Werten – und zurückgegeben.

```
...
@NamedQuery(
  name="findAllDMDiagrammValues",
  query=  " select new dipl.modern.db.pojo.DMDiagrammValue(" +
      "  substring(t.durchfuehrungsDate, 1, 7), " +
      "  sum(case when t.betrag > 0 then t.betrag else 0 end), " +
      "  sum(case when t.betrag < 0 then t.betrag else 0 end)) " +
      " from Transaktion t " +
      " group by substring(t.durchfuehrungsDate, 1, 7)" +
      " order by substring(t.durchfuehrungsDate, 1, 7) asc ")
...
```

Quellcode 4.10: DM Query für Summenanzeige

Das Ergebnis aus dieser Abfrage muss nur wie in Kapitel 2.2.4.1 beschrieben an die „Series„ übergeben werden, die sich in einer „ObservableList“ befinden. Diese Liste wird anschließend beim Erstellen des „BarChart„ übergeben. Dank der Eigenschaften der „ObservableList“ reicht es beim Aktualisieren einfach die Werte dieser Liste auszutauschen.

Lediglich das vertikale Scrollen in der Anzeige muss noch selbst entwickelt werden. Dafür wird einfach ein „AnchorPane“ verwendet, an dem man das Diagramm nur oben, links und unten befestigt. Somit läuft das Diagramm – falls notwendig – rechts aus dem Fenster hinaus. Damit man hier das vertikale Scrollen ermöglicht, wird wie in Quellcode 4.11 beschrieben bei einem Maus-Klick die Position des linken Ankers verschoben (analog zur Lösung aus der DO-Entwicklung, lediglich mit Java-FX Handlern).

```
...
addEventHandler(MouseEvent.MOUSE_DRAGGED, new EventHandler<MouseEvent>() {
  private double oldMouseX = 0;
  @Override
  public void handle(MouseEvent event) {
    double delta = event.getX() - oldMouseX;
    double oldBarChartX = getLeftAnchor(barChart);
    if(delta >= 50)
      delta = 50;
    else if(delta <= -50)
      delta = -50;
    Logger.getLogger("DEFAULT").debug(
      "Mausposition: " + event.getX() +
      ", Delta: " + delta +
      ", alte-Diagrammposition: " + oldBarChartX);
    setLeftAnchor(barChart, oldBarChartX + delta);
    oldMouseX = event.getX();
  }
});
...
```

Quellcode 4.11: DM Query für Summenanzeige

Abgesehen davon verlief die Entwicklung der Diagramm-Anzeige völlig problemlos und erstaunlich schnell.

4.3.3 Summen-Panel

4.3.3.1 DO-Entwicklung

Hier wird auf keine Framework-Komponenten zugegriffen. Es entstand die Klasse **DO-SummenPanel**, welche wie in Quellcode 4.12 beschrieben lediglich ein SQL-Statement durchführt, welches eine Auflistung des Kontostandes je Konto – und den gesamten Vermögensstand – anzeigt.

```
...
HashMap<String, String> labelAndVal = new HashMap<String, String>();
double sumGesamt = 0.0;

Statement st = DODatabaseConnection.getInstance().createStatement();
st.executeQuery("select k.name, sum(t.betrag) " +
  " from transaktion t, konto k " +
  " where k.id = t.fkonto " +
  " group by k.name ");

ResultSet rs = st.getResultSet();
while(rs.next()){
  String name = rs.getString(1);
  double sum = rs.getDouble(2);
  sumGesamt += sum;
  labelAndVal.put(name, format.format(sum));
}
...
```

Quellcode 4.12: DO Query für Summenanzeige

Abgesehen davon musste noch implementiert werden, dass sich die Anzeige nach mögliche Kontoänderungen aktualisiert.

4.3.3.2 DM-Entwicklung

Das Panel **DMSummenPanel** konnte ohne weitere Probleme schnell erstellt werden. Um die Modularität und Übersicht zu wahren, wurde hierfür ein eigenes Datenobjekt **DMKontostand** erstellt, welches die beiden String-Properties **kontoname** und **kontosumme** enthält. Java FX bietet hier die Möglichkeit die Ergebnisdaten bei einer Query direkt in ein übergebenes Datenobjekt zu befüllen. Hierfür wurde in Quellcode 4.13 die „NamedQuery“ (siehe Kapitel 2.1.5.2) definiert.

```
...
@NamedQuery(
  name="findAllKontostand",
  query=  " select new dipl.modern.db.pojo.DMKontostand( " +
      " t.fkonto.name, sum(t.betrag)) " +
      " from Transaktion t " +
      " group by t.fkonto.name "),
...
```

Quellcode 4.13: DM Query für Summenanzeige

Quellcode 4.14 zeigt wie diese Query direkt eine Liste des Datentypes „DMKontostand“ liefert.

```
...
private List<DMKontostand> kontostandList kontostandList =
  DMDatabaseConnection.getIntance().getEntityManager().createNamedQuery(
    "findAllKontostand", DMKontostand.class
  ).getResultList();
...
```

Quellcode 4.14: DM intelligenter Query-Output für Summenanzeige

Mit diesem ideal für diesen Anwendungsfall geeigentem Output konnte die Anzeige ohne Probleme mittels dem „GridPane“ (siehe Kapitel 2.2.5.3) entwickelt werden.

4.3.4 Zusammenführung der Komponenten

4.3.4.1 DO-Entwicklung

Das Zusammenführen der einzelnen Komponenten in der Klasse **DOApplikation** erfolgt äußerst einfach und unkompliziert. Da jegliche Kommunikation zwischen den Panels bereits entwickelt wurde, muss hier nur noch die Positionierung mittels „SpringLayout“ vorgenommen werden. Auch die Funktionalität, dass das Schließen des Fensters nur mit dem Drücken auf „x“ erlaubt ist, falls die Tabelle sich nicht im Bearbeitungszustand befindet, muss hier implementiert werden. Abbildung 4.1 zeigt die fertige Anzeige der DO-Applikation.

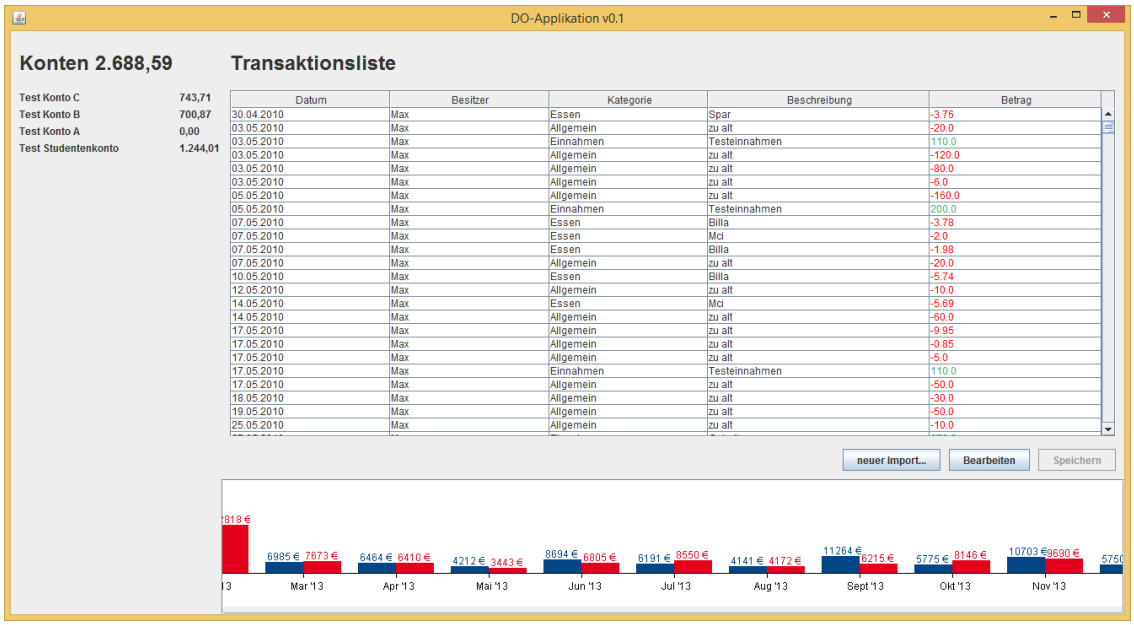


Abbildung 4.1: fertige DO-Applikation

4.3.4.2 DM-Entwicklung

Auch hier gab es beim Zusammenführen keine Probleme. Die Klasse **DMApplikation** positioniert die einzeln erstellten Diagramme mittels „AnchorPane“. Die Implementierung, dass das Fenster im Bearbeitungsmodus nicht terminiert werden darf, ist analog zur DO-Implementation über einen Listener auf das Fenster realisiert. Abbildung 4.2 zeigt die fertige Anzeige der DM-Applikation.

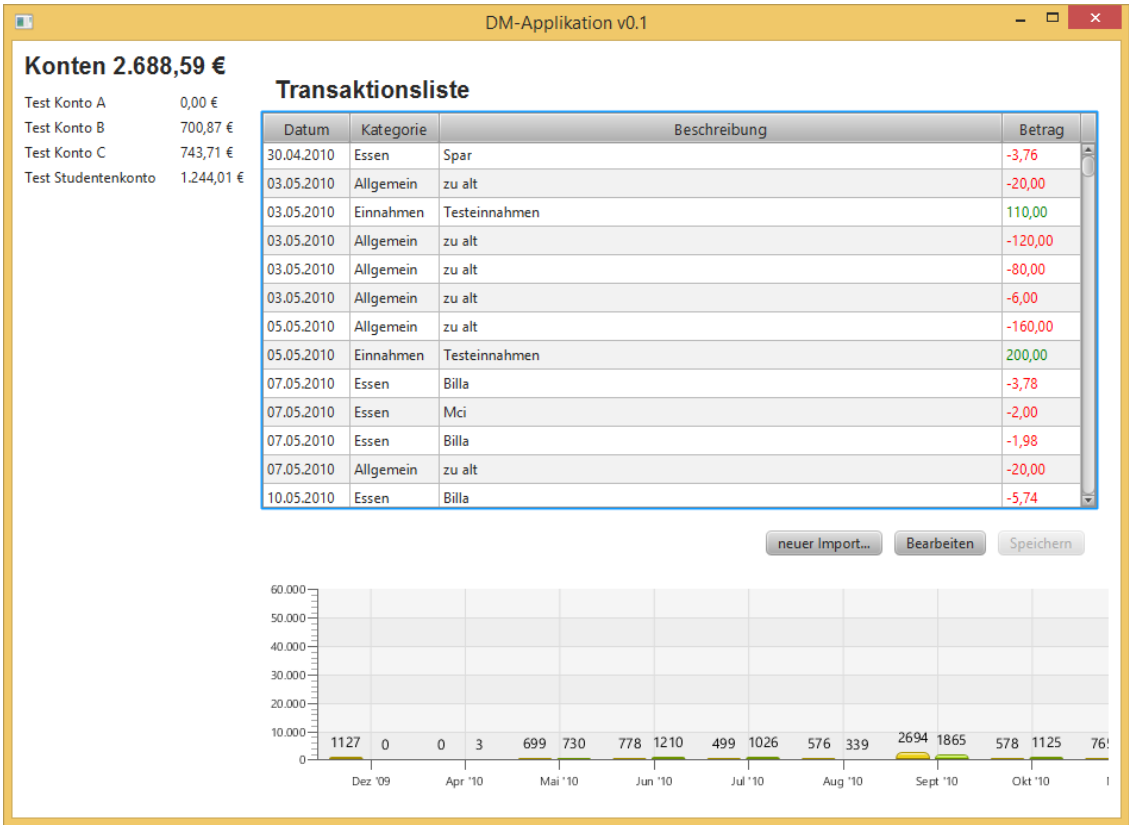


Abbildung 4.2: fertige DM-Applikation

4.4 Erweiterung der Applikation

Hier sollen beide Applikationen kurz erweitert werden, damit anschließend ermittelt werden kann, wie sich bei der jeweiligen Implementierung der Aufwand verhält. Hierfür gibt es zwei Änderungswünsche:

1. Die Spalte „Betrag“ der Tabelle „Transaktion“ wird umbenannt in „Transaktionsbetrag“. Diese Erweiterung betrifft die Datenbankabstraktion.
2. Die Summen-Anzeige der Konten soll alphabetisch sortiert werden. Hier soll nicht mittels dem SQL-Befehl „order by“ vorgegangen werden, sondern im Java sortiert werden.

4.4.1 Aufwand bei Datenbankänderung

4.4.1.1 DO-Entwicklung

Der Aufwand verhält sich hier in einem unberechenbaren Rahmen. Die Implementierung hat sich als äußerst tükisch erwiesen, da es zuerst den Eindruck macht, dass alles schnell erweitert werden kann. Erst beim Testen fällt auf an wie vielen Stellen der Quellcode noch zusätzlich manipuliert werden muss.

Es muss beim Update-Statement im „DOTransaktionHome“ der Variablenname „betrag“ ausgetauscht werden. Abgesehen davon musste in der Home-Klasse nichts gemacht werden, da sowohl beim Einlesen und beim Erzeugen nicht auf den Spaltennamen, sondern die Spaltenposition in der Datenbank zugegriffen wird. Allerdings kam es beim Starten der Applikation zu einer SQL-Exception, denn im „DODiagrammPanel“ werden die Werte nicht über die POJOs, sondern über ein direktes SQL-Statement ermittelt, in dem auch die umbenannte Spalte vorkommt. Wenn man auch in dieser Query den Spaltennamen austauscht, startet die Applikation wieder einwandfrei. Auch im „DOSummenPanel“ – in dem eine direkte SQL-Query aufgerufen wird – kam es zum selben Fehler. Auch hier musste der neue Spaltenname eingetragen werden.

Der Erweiterungsaufwand beläuft sich somit auf fünf Zeilen in drei Klassen.

4.4.1.2 DM-Entwicklung

Hier muss lediglich im POJO „DMTransaktion“, bei der richtigen Spalte der Annotation „@Column“, der neue Name der Spalte übergeben werden. Da die Queries in den beiden Panels „DMDiagrammPanel“, und „DMSummenPanel“ in JPQL geschrieben wurden, suchen diese über die Java-POJOs. Hier hat sich der Name des Properties „betrag“ allerdings nicht geändert. Darum funktionieren die Queries weiterhin.

Der Erweiterungsaufwand beläuft sich damit auf eine Zeile in einer Klasse.

4.4.2 Aufwand bei alphabetischer Sortierung der Komponenten

4.4.2.1 DO-Entwiclung

Hier muss die Sortierung manuell vorgenommen werden, da für „HashMaps“ keine Methoden zur Sortierung angeboten werden. Hierfür kann man sich das „KeySet“ als Array zurückgeben lassen, und mittels der Java-Standardmethode „Arrays.sort()“ die Sortierung vornehmen. Anschließend muss man eine „LinkedHashMap“ – also eine sortierte „HashMap“ – in der Reihenfolge des sortierten Arrays befüllen. Quellcode 4.15 beschreibt den gesamten Sortiervorgang.

```
...
  // LabelAndVal beinhaltet die unsortierten KONTEN mit dem aktuellen KONTOSTAND
  String[] keys = labelAndVal.keySet().toArray(new String[labelAndVal.size()]);
  Arrays.sort(keys);
  HashMap<String, String> sortedMap = new LinkedHashMap<String, String>();
  for(String key:keys)
    sortedMap.put(key, labelAndVal.get(key));
  labelAndVal = sortedMap;
...
```

Quellcode 4.15: DO Sortierung der Konten

Somit beläuft sich der Erweiterungsaufwand hier auf sechs Zeilen.

4.4.2.2 DM-Entwicklung

Da hier der Rückgabewert der Abfrage bereits eine Liste mit dem Datentyp „DMKontostand“ ist, und für Listen die Sortiermethode „Collections.sort()“ zur Verfügung gestellt wird, muss lediglich ein „Comperator“ definiert werden, der beschreibt anhand welcher Daten sortiert werden soll. Da hier nach dem Property „kontoname“ sortiert werden soll, folgt die Implementierung laut Quellcode 4.16.

```
...
  // KONTOSTANDLIST enthaelt ermittelte Konten und deren Staende
  Collections.sort(kontostandList, new Comparator<DMKontostand>() {
    @Override
    public int compare(DMKontostand dk1, DMKontostand dk2) {
      return dk1.getKontoname().compareTo(dk2.getKontoname());
    }
  });
...
```

Quellcode 4.16: DM Sortierung der Konten

Der Erweiterungsaufwand beläuft sich daher hier (exkl. Klammerung) auf drei Zeilen.

5 Technologievergleich

Hier werden die Erkenntnisse der Entwicklung der Testapplikation aus Kapitel 4 schrittweise analysiert. Als Vergleichsparameter werden die in Kapitel 3.1 definierten Punkte verwendet.

5.1 Erkenntnisse Entwicklungsvergleich

Zur Auswertung der Einarbeitungs- und Entwicklungszeiten wird die Zeitaufzeichnung aus Anhang 3 verwendet. Zur Qualitätsbestimmung des Quellcodes wird das gängige Analysetool SonarQuebe[28] verwendet. Die Qualität der Entwicklung lässt sich anhand bestimmter vorgegebener Parameter messen. Alles in allem kann man sagen: desto mehr Zeilen es im Quellcode gibt, desto höher ist die Gefahr, dass sich darin ein Fehler verbirgt.

Außerdem ist es auch wichtig, dass der Code leserlich bleibt, um einen neuem Entwickler die Chance zu geben eine Klasse rasch zu erweitern. Hierfür gilt es die Komplexität möglichst niedrig zu halten. Als Vergleichsparameter wurden hierfür folgende Eigenschaften gewählt.

- Statement-Anzahl
- Methodenanzahl
- Zeilenanzahl
- Klassenanzahl
- Komplexitätsgrad[29]

5.1.1 Datenbankabstraktion

5.1.1.1 Beschreibung der Entwicklung

DO-Entwicklung

Hier musste alles eigens entwickelt werden – vom Systemkonzept bis zu der Datenbankanbindung. Der Entwicklungsaufwand war bereits bei wenigen Datenbanktabellen sehr groß und somit stieg auch die Fehleranfälligkeit. Alles in allem war die Implementierung zwar komplett flexibel – da man sich nicht mit den Einschränkungen einer Technologie

[28] siehe http://www.sonarqube.org/
[29] wird durch SonarQuebe vergeben

konfrontieren musste – allerdings erwies sich die angebliche Starrheit der modernen Technologien eher als Mythos.

DM-Entwicklung

Nach einer notwendigen Einlesezeit konnte JPA ohne Probleme genutzt werden. Die Implementierung war einfach und schnell erledigt. Die Technologie bietet dermaßen viele Möglichkeiten, dass kein entwicklungstechnischer Nachteil erkannt werden konnte. Ganz im Gegenteil bietet JPA viel mehr Möglichkeiten, die aus Zeitgründen in der eigens entwickelten Lösung nicht enthalten sind (wie zum Beispiel die automatisierte Erkennung von Änderungen in den POJOs).

5.1.1.2 Quantität der Entwicklung

Die sinnvolle Einarbeitungszeit in JPA mit der Entwicklung kleiner Testapplikationen (im Gegensatz zum raschen Überfliegen) nahm einen beträchtlichen Zeitaufwand in Anspruch. Die Zeitdauern in Tabelle 5.1 wurden aus der geführten Zeitaufzeichnung (siehe Anhang 3) ermittelt.

	eigene Entwicklung	**Entwicklung mittels JPA**
Einarbeitungszeit	0 Stunden	42 Stunden
Entwicklungszeit	11,5 Stunden	3 Stunden

Tabelle 5.1: Vergleich der Entwicklungszeit bei der Datenbankabstraktion

Bereits bei der Entwicklung der ersten Applikation konnte die Datenabstraktion beinahe vier Mal schneller erstellt werden als bei einer eigens entwickelten Methode.

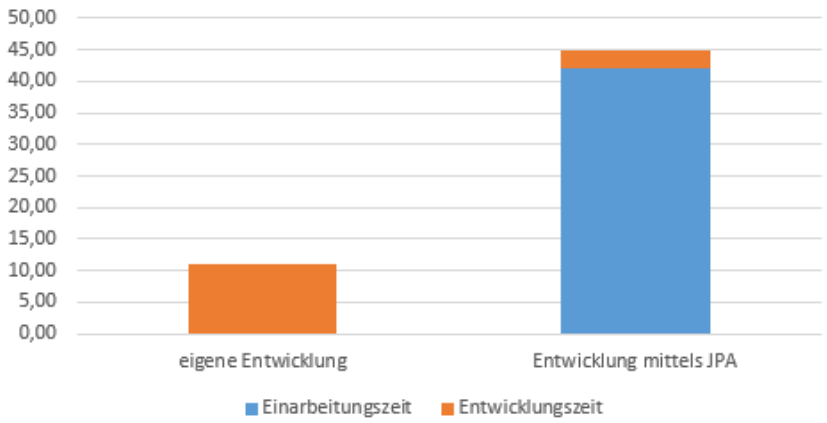


Abbildung 5.1: Vergleich der Entwicklungszeit bei der Datenbankabstraktion

5.1.1.3 Qualität der Entwicklung

Hier unterscheiden sich die beiden Entwicklungen stark. Während in JPA auf Standard-Methoden zugegriffen wird und die Technologie den Großteil der Arbeit übernimmt, muss in der eigens entwickelten Methode alles selbst implementiert werden. Dadurch steigt die Anzahl der Zeilen, die Fehleranfälligkeit und auch die Komplexität.

	eigene Entwicklung	Entwicklung mittels JPA
Methoden	44	20
Statements	243	11
Klassen	11	7
Zeilen	619	391
Komplexität	71	33

Tabelle 5.2: Vergleich der Qualität bei der Datenbankabstraktion

In Tabelle 5.2 und Abbildung 5.2 sieht man auf den ersten Blick, dass JPA den Aufwand und die Komplexität in allen Punkten circa halbiert.

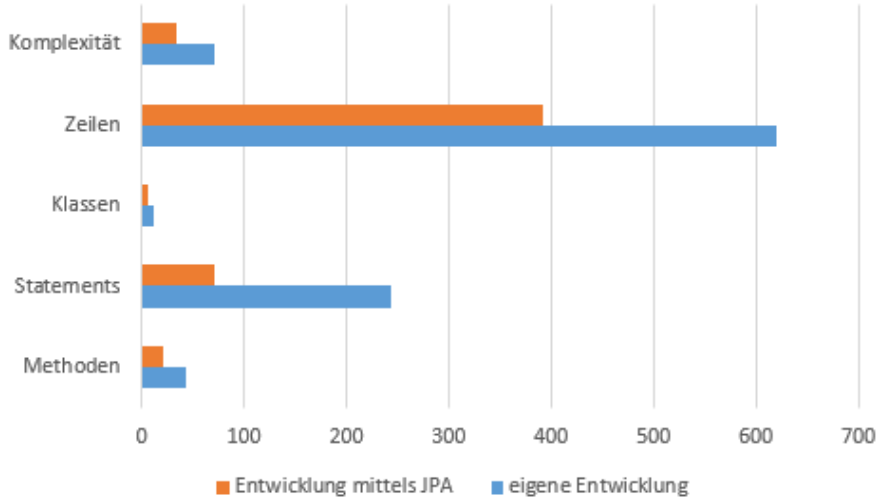


Abbildung 5.2: Vergleich der Qualität bei der Datenbankabstraktion

5.1.1.4 Analysequalität

Dieser Punkt lässt sich schwer mit Parametern festlegen. Rein subjektiv ist in beiden Fällen das Debuggen und Analysieren sehr gut möglich.

In der eigenen Lösung hat man Zugriff auf alle Quellcode-Dateien und kann hier bequem arbeiten. Die Fehlermeldungen geben immer sinnvolle Rückmeldungen. Auch die eigenen Logging-Meldungen reichen für ein kleines Projekt aus.

Im Gegensatz dazu ist JPA sehr gut dokumentiert. Die Fehlermeldungen sind sehr sinnvoll und geben Anhaltspunkte, warum es zu dem Fehler gekommen sein kann. JPA verwendet den Java-Logger. Die eigenen Log-Meldungen durch LOG4J sind auch sehr gut und sinnvoll.

5.1.1.5 Fazit

Nach den Erfahrungen bei der Entwicklung, Anwendung, Fehlersuche und der anschließenden Analyse wird dringlich geraten – falls möglich – die Einarbeitungszeiten in Kauf zu nehmen und auf JPA umzusteigen. Schon nach mehreren Wochen amortisiert sich diese Investition. Die Technologie ist sehr einfach zu verwenden, der Code bleibt durch die Annotations immer übersichtlich. Die Möglichkeiten sind sehr flexibel. Es wurde keine Schwachstelle oder Einschränkung in der Technologie gefunden.

Die Technologie ist in jeglicher Hinsicht der eigens entwickelten Lösung überlegen. Auch die direkte Abfrage mittels SQL in den Panels „SummenPanel“ und „DiagrammPanel“ ist in JPA – dadurch, dass JPQL verwendet wird – einem reinen JDBC-Statement überlegen, wie sich bei der Erweiterung der Applikation gezeigt hat.

Vom entwicklungstechnischen Standpunkt ist eine Verwendung von JPA stark zu empfehlen. Ob es bei JPA zu Performance-Einbußen oder Overhead kommt, wird in Kapitel 5.2.1 behandelt.

5.1.2 Tabelle Transaktionsliste

5.1.2.1 Beschreibung der Entwicklung

DO-Entwicklung

Hier mussten extra Framework-Komponenten entwickelt werden, da die von Java-Swing zur Verfügung gestellten Klassen zwar viele Möglichkeiten bieten, allerdings mit viel Implementierungsaufwand verbunden sind. Abgesehen davon hat alles wie gewohnt einwandfrei funktioniert.

DM-Entwicklung

Hier kann gänzlich auf die eigens entwickelten Framework-Komponenten verzichtet werden, da Java-FX hierfür bereits den „TableViewBuilder“ zur Verfügung stellt. Mit diesem kann mit äußerst geringem Aufwand eine komplexe Tabelle aufgebaut werden. Allerdings merkt man schnell, dass Java-FX eine junge Technologie mit ihren Fehlern ist.

Man stößt bei der Entwicklung auf Bugs, die Dokumentation ist teilweise noch schwach und es sind im Internet kaum Entwicklungsbeispiele zu finden.

5.1.2.2 Quantität der Entwicklung

Hier hält sich der Einarbeitungsaufwand bei Java FX mit 4,5 Stunden stark in Grenzen, da das Thema sehr spezifisch ist. Bei der Entwicklung hingegen konnte mittels Java FX bereits in 4,5 Stunden eine komplett funktionsfähige Tabelle erstellt werden.

Im Vergleich dazu dauerte die Entwicklung – mit der Erstellung aller notwendigen Framework-Klassen – der Java-Swing Tabelle 8,5 Stunden. Die Zeitangaben aus Tabelle 5.3 beziehen sich auf die Zeitaufzeichnung aus Anhang 3.

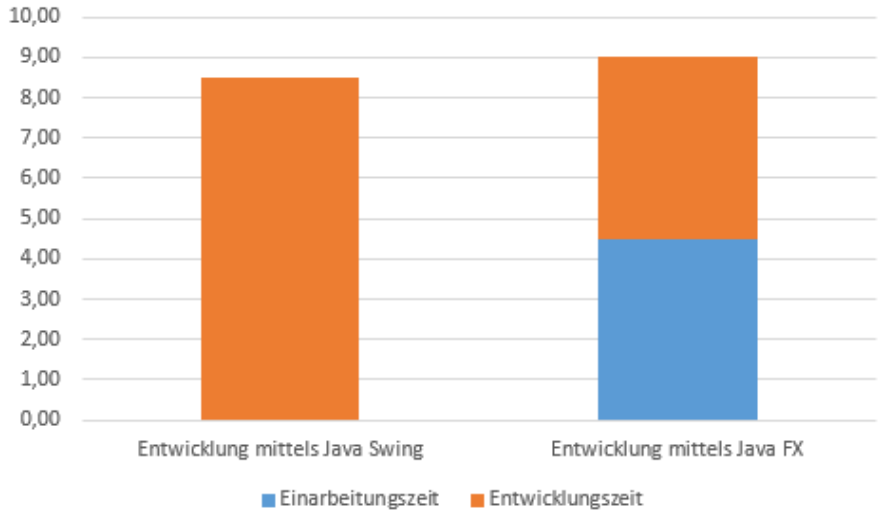


Abbildung 5.3: Vergleich der Entwicklungszeit bei der Tabellenentwicklung

5.1.2.3 Qualität der Entwicklung

Durch die komplizierte Entwicklung der Java-Swing Tabelle gehen auch hier die Qualitätsparameter stark auseinander wie die Abbildung 5.4 zeigt.

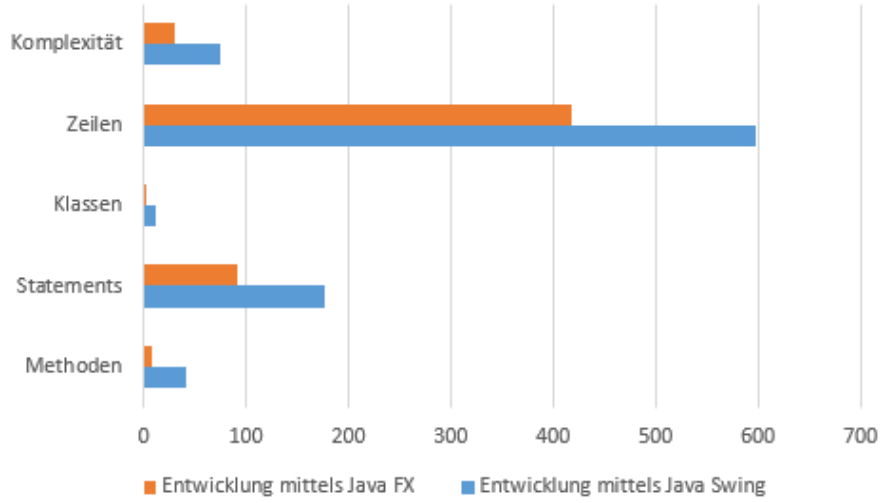


Abbildung 5.4: Vergleich der Qualität bei der Tabellenentwicklung

5.1.2.4 Analysequalität

Im Bereich GUI ist die Analyse meistens schwer, da ein möglicher Fehler in der Anzeige keineswegs ein Fehler im Code heißen muss. In Java-Swing kann man sehr viel konfigurieren. Hier funktioniert alles und wenn es zu einem Fehler kommt, liegt es meistens am Entwickler. Im Vergleich dazu sind in Java FX richtige Bugs aufgefallen. Diese kosten unnötige Zeit und machen die Entwicklung anstrengend. Hier lässt sich erkennen, dass die Technologie noch in den Anfängen steckt.

5.1.2.5 Fazit

Trotz der (noch) existierenden Schwächen von Java FX konnte die Implementierung in der Hälfte der Zeit der Java-Swing Tabelle erstellt werden. Der kurze Einarbeitungsaufwand zahlt sich voll und ganz aus. Ein Umstieg auf Java-FX ist in diesem Bereich sehr ratsam.

5.1.3 Diagramm-Panel

5.1.3.1 Beschreibung der Entwicklung

DO-Entwicklung

Hier musste eine eigene Diagramm-Logik mithilfe des „Graphics“-Objekts aus Java-Swing erzeugt werden. Diese Entwicklung ist zwar komplett dynamisch, allerdings vergleichsweise aufwendig.

DM-Entwicklung

In Java FX gibt es bereits eine sinnvolle Diagramm-Technologie, die ähnlich wie die anderen GUI-Komponenten sehr leicht zu konfigurieren und schon fast intuitiv zum anwenden ist.

5.1.3.2 Quantität der Entwicklung

Auch hier zeigt die Abbildung 5.5, dass man bei Java FX die Einarbeitungszeit den Großteil der Dauer in Anspruch nimmt. Bei einem größeren Projekt würde bereits nach kurzer Zeit die einfache Implementierbarkeit von Java FX die verlorene Zeit durch die Einarbeitung gut machen.

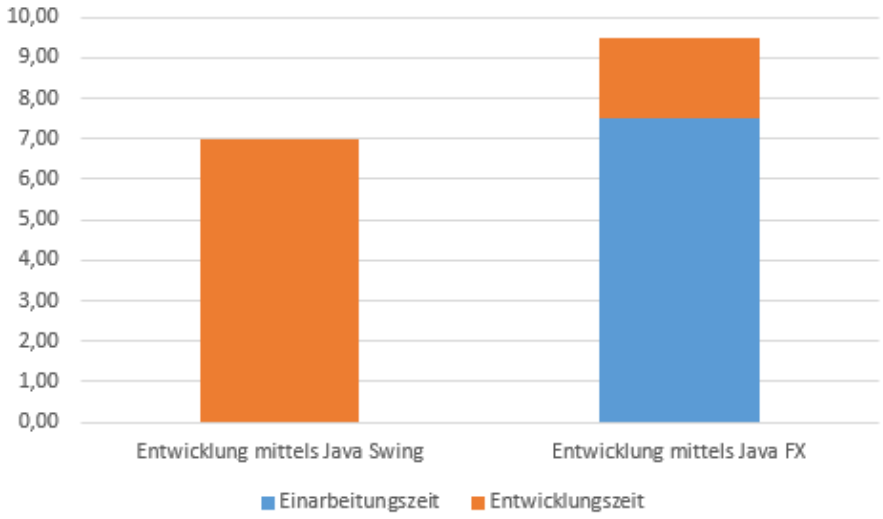


Abbildung 5.5: Vergleich der Entwicklungszeit bei der Diagrammentwicklung

5.1.3.3 Qualität der Entwicklung

Analog zu den anderen Punkten ist auch hier Java FX sehr einfach zu implementieren. Die Entwicklung bedarf wie Abbildung 5.6 zeigt gewohnt wenig Aufwand.

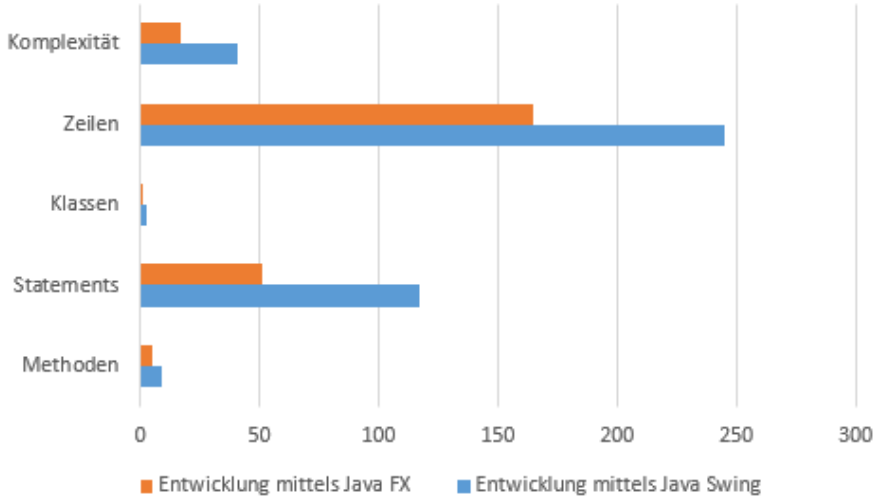


Abbildung 5.6: Vergleich der Qualität bei der Diagrammentwicklung

5.1.3.4 Analysequalität

Hier kann man bei beiden Lösungen in die Tiefe gehen und einfach Fehler suchen und Anpassungen durchführen. Im Gegensatz zur Tabellen-Implementierung scheint Java-FX in diesem Bereich erwachsen und fehlerfrei zu sein.

5.1.3.5 Fazit

Es bestätigt sich hier nur die Aussage der letzten Vergleiche. Durch die neue Technologie sinkt der Aufwand nach der Einarbeitungsphase stark, die Qualität hingegen steigt. Dadurch, dass die Implementierung in diesem Fall noch dazu erwachsen erscheint, gibt es keinen Grund – wenn es um Diagramme geht – nicht auf Java-FX umzusteigen.

5.1.4 Summen-Panel

5.1.4.1 Beschreibung der Entwicklung

DO-Entwicklung

Hier konnte das Panel relativ einfach aufgezogen werden. Da in diesem Fall auf keine großartigen Funktionen zugegriffen wurde, gibt es hier nichts weiteres zu beschreiben.

DM-Entwicklung

Auch hier wurden keine großartigen Funktionen von Java FX verwendet. Allerdings ist die Implementierung durch das Zusammenspiel mit den Java-JPA-POJOs etwas übersichtlicher.

5.1.4.2 Quantität der Entwicklung

Dieses Panel ist das Erste bei welchem, aufgrund der Erfahrung aus den vorhergehenden Panels und des allgemeinen Einarbeitens, keine Einarbeitungszeit bei Java-FX entstanden ist. Allerdings ist es auch das Erste, welches mit reinen Standard-Methoden von Java-Swing auskommt. Wie Abbildung 5.7 zeigt folgt daraus, dass aufgrund der Erfahrungen mit Java-Swing einfache Panels noch um einiges schneller entwickelt werden können als mit der neuen Technologie.

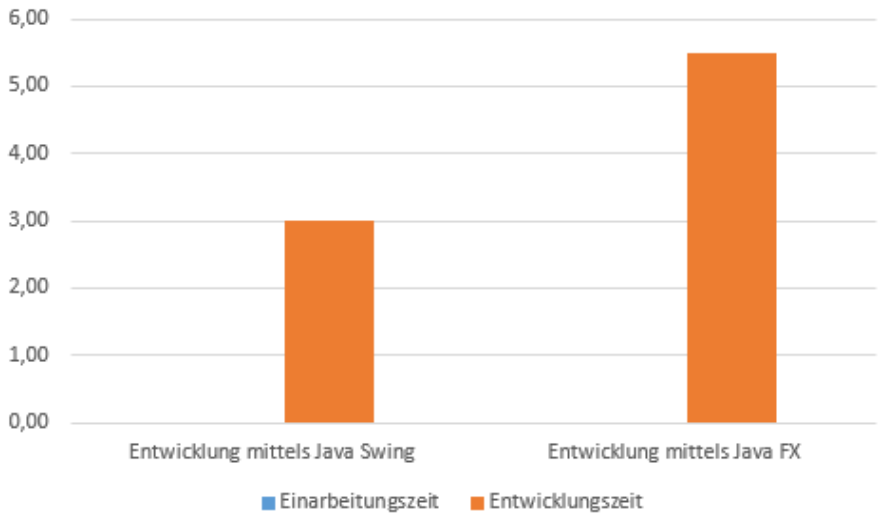


Abbildung 5.7: Vergleich der Entwicklungszeit bei der Summen-Anzeigeentwicklung

5.1.4.3 Qualität der Entwicklung

Obwohl die Entwicklung in Java-Swing schneller ging, setzt sich in Abbildung 5.8 bei den Qualitätsrichtlinien das gewohnte Bild durch. Die neuere Technologie ermöglicht ein einfacheres und weniger komplexes Entwickeln.

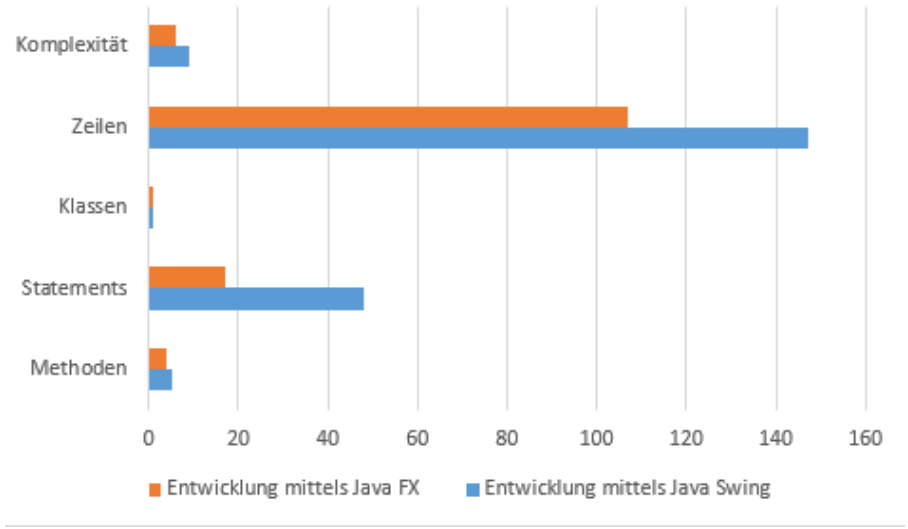


Abbildung 5.8: Vergleich der Qualität bei der Summen-Anzeigeentwicklung

5.1.4.4 Analysequalität

Auch hier unterscheiden sich die Qualität des Debuggings und des Loggings nicht stark von einander. Sowohl der eigens entwickelte Logger als auch Log4J liefern genug Informationen. Da bei diesem Panel nicht viele technologiespezifische Eigenschaften verwendet werden, ist es schwierig hier einen Unterschied festzustellen.

5.1.4.5 Fazit

Bei einfachen Anwendungen ist es ratsamer die Technologie zu verwenden in der man routinierter ist. Wenn man allerdings an einer größeren Applikation arbeitet, wird der Spalt zwischen diesen Implementierungszeiten immer kleiner, bis er irgendwann verschwindet. Bei größeren Projekten hingegen ist die Qualität das Wichtigste und daher ist auch hier die modernere Technologie zu empfehlen.

5.1.5 gesamte Applikation

5.1.5.1 Beschreibung der Entwicklung

DO-Entwicklung

Das Zusammenführen der Applikation war hier kein Problem, da die gesamte Logik bereits in den einzelnen Panels abgebildet wurde. Die einzelnen Panels der gesamten Applikation kommunizieren fehlerfrei miteinander. In früheren Java Versionen kam es hin und wieder zu Problemen bei der Schachtelung verschiedener Layout-Manager. Die-

se Probleme konnten hier nicht nachvollzogen werden, obwohl mehrere verschiedene Layout-Manager zum Einsatz kamen.

DM-Entwicklung

Auch hier hat es beim Zusammenführen keine Probleme gegeben. Alle Layouts harmonieren miteinander. Der Code bleibt wie gewohnt schlank und übersichtlich.

5.1.5.2 Quantität der Entwicklung

Erst über die gesamte Applikation betrachtet, fällt auf dass der Einarbeitungsaufwand bei den modernen Technologien doch immens war (siehe Abbildung 5.3). Es gibt Einarbeitungsaufwände, die global für die Technologie gelten und somit bei den einzelnen Statistiken der Panels nicht enthalten sind.

	eigene Entwicklung	**Entwicklung mittels JPA und Java FX**
Einarbeitungszeit	0 Stunden	71 Stunden
Entwicklungszeit	37 Stunden	17,5 Stunden

Tabelle 5.3: Vergleich der Entwicklungszeit über die gesamte Applikation

Allerdings fällt auch auf, dass die Entwicklungszeit der modernen Applikation anschließend nur bei 47 Prozent der Arbeitszeit der anderen Applikation liegt.

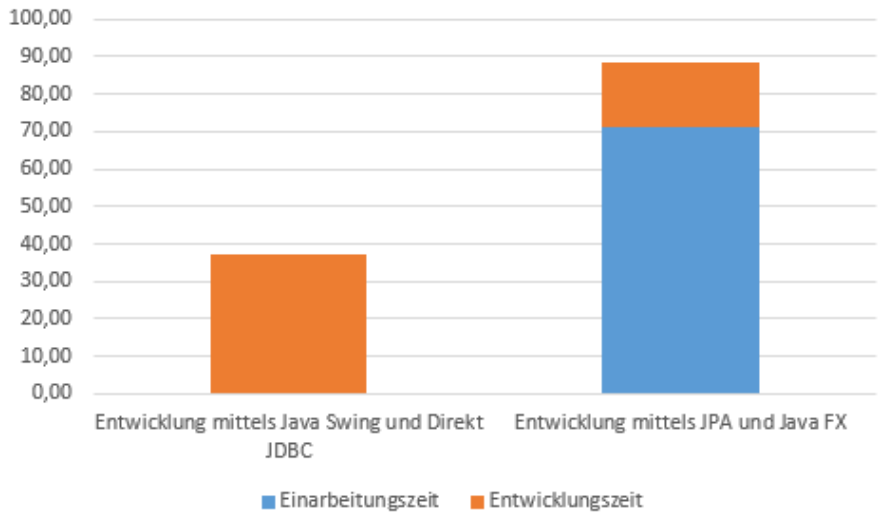


Abbildung 5.9: Vergleich der Entwicklungszeit bei der gesamten Applikation

5.1.5.3 Qualität der Entwicklung

Die Erkenntnis die bei den einzelnen Panels gemacht wurde zieht sich auch hier durch. Wie Tabelle 5.4 zeigt ist die moderne Applikation aus Sicht der Qualitätsrichtlinien der herkömmlichen Entwicklung in jeglicher Hinsicht stark überlegen.

	eigene Entwicklung	**Entwicklung mittels JPA und Java FX**
Methoden	111	41
Statements	650	273
Klassen	28	12
Zeilen	1554	1026
Komplexität	226	97

Tabelle 5.4: Vergleich der Qualität bei der gesamten Applikation

Im ausschlaggebenden Punkt – der Komplexitätsbewertung – konnte sogar ein Wert erzielt werden, der unter 50 Prozent von jedem der herkömmlichen Applikation liegt.

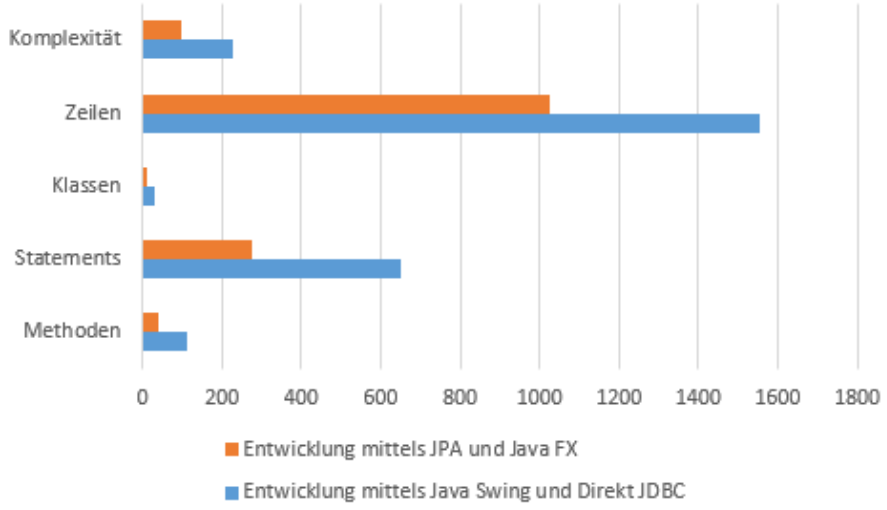


Abbildung 5.10: Vergleich der Qualität bei der gesamten Applikation

5.1.5.4 Analysequalität

Auch über die gesamte Applikation gesehen, fällt es schwer hier ein Urteil zu treffen. Bei beiden Applikationen kann man das Logging, als auch das Troubleshooting, als ausreichend gut bezeichnen. Es sollte lediglich erwähnt werden, dass bei der herkömmlichen Applikation die Verantwortung für sinnvolle Log-Meldungen und Fehlermeldungen eher beim Entwickler liegen. Allerdings kann man hier durch ein ordentliches Konzept problemlos mit den modernen Technologien mithalten.

5.1.5.5 Fazit

Der Einarbeitungsaufwand über die gesamte Applikation gesehen ist doch immens und nimmt über 80 Prozent der Arbeitszeit an der modernen Applikation in Anspruch. Die Entwicklungszeit bei der modernen Applikation beträgt hingegen nur 47 Prozent der herkömmlichen Applikation, was auch ein beachtliches Ergebnis ist. Es zeigt sich, dass über eine kleine Applikation doch ein Unterschied von über 40 Stunden in der gesamten Arbeitszeit entsteht. Hinsichtlich der Qualität konnten die modernen Technologien in jedem Punkt überzeugen. Allerdings spielt der Kostenfaktor in der Realwirtschaft meist eine höhere Rolle.

Da bei einer kleinen Applikation durch die Verwendung herkömmlicher Technologien in der Hälfte der Arbeitszeit ein Ergebnis erzielt werden kann ist aus rein wirtschaftlicher Sicht die Verwendung der herkömmlichen Technologien anzuraten. Das gilt natürlich nur für einmalige Aufträge, denn sobald eine Applikation auch gewartet order erweitert werden soll, entsteht durch die herkömmlichen Technologien wieder ein immenser Mehraufwand. Handelt es sich allerdings um ein größeres Projekt, holt man ab einem gewissen Punkt den Aufwand für die Einarbeitungszeit auf. Für solche Szenarien gibt es in der Kostenrechnung den „Break-Even-Punkt“[30]. Dieser Punkt kann abstrahiert auch in diesem Fall zum Einsatz kommen.

Wenn man die Einarbeitungszeit für diese beiden Technologien – mit denen man problemlos eine Rich-Client-Applikation aufziehen kann – als konstante t_{ao} mit 85 Stunden (71 Stunden laut Zeitaufzeichnung plus 20 Prozent Risikozuschlag) bewertet, und die Entwicklungszeit der modernen Applikation anhand der Erkenntnisse bei der Testapplikation mit 50 Prozent von t_{ao} bewertet – wobei t_{ao} für die Arbeitszeit der herkömmlichen Applikation steht – dann folgt daraus, dass sich aus wirtschaftlicher Sicht die Verwendung der neuen Technologien ab einem klassischem Entwicklungsaufwand von 170 Stunden auszahlt.

$$\frac{t_{ao}}{2} + \underbrace{85}_{t_{em}} = t_{am}$$

- t_{ao} Arbeitszeit bei eigener Entwicklung mittels Java Swing und JDBC (in Stunden)
- t_{em} Einarbeitungszeit JPA und Java FX (in Stunden)
- t_{am} Arbeitszeit mit JPA und Java FX (in Stunden)

Sobald die Entwicklung einer Applikation – bei der mit Java Swing und Driect-JDBC geplant wird – auf über 170 Stunden je Entwickler geschätzt wird, sollte man wie in Abbildung 5.11 dargestellt aus rein wirtschaftlicher Sicht auf die modernen Technologien

[30] Punkt an dem sich die vorangegangenen Investitionen – in diesem Fall die Einarbeitungszeit – auszahlen

wechseln.

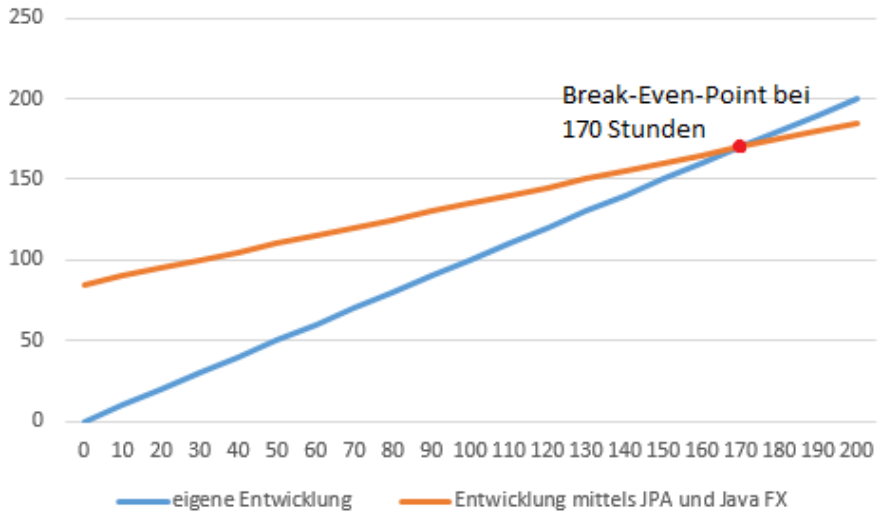


Abbildung 5.11: Amortisation der Einarbeitungszeit in JPA und Java FX

Da man heutzutage damit rechnen muss, dass jede ausgelieferte Software überarbeitet oder erweitert werden muss, ist aufgrund der geringen Entwicklungszeit bei neuen Projekten in jedem Fall ein Umstieg auf die modernen Technologien ratsam.

5.2 Erkenntnisse Laufzeitvergleich

Hier werden die Performance-Werte in zwei wesentlichen Punkten verglichen:

- **Rechenzeit:** wird mittels Logging-Einträgen berechnet und ausgegeben
- **Arbeitsspeicherverbrauch:** wird mittels dem Analysetool JVisualVM[31] ermittelt

Bei den Oberflächentests werden die Unterschiede in der Usability gesucht, beschrieben und anschließend bewertet.

5.2.1 Datenbankabstraktion

5.2.1.1 Performance

Es wurden drei Tests durchgeführt, um die Laufzeit anhand verschiedener Punkte zu messen.

- **Verbindungsaufbau:** Es wird verglichen, wie lange die beiden Datenbankanbindungen brauchen, um eine Verbindung zur Datenbank zu initialisieren.

[31] http://docs.oracle.com/javase/7/docs/technotes/guides/visualvm/

- **Summen-Abfrage:** Es wird eine Abfrage erstellt, die die Summe über alle Transaktionen ausgibt. Während die SQL-Abfrage gänzlich auf der Datenbank durchgeführt wird und nur eine Zeile zurück gibt, wird vermutet, dass hier via JPQL – welches über die Objekte hinweg arbeitet – erheblich mehr Aufwand entsteht.
- **Listen-Abfrage:** Es sollen alle Transaktionen als Liste mit Datenobjekten ermittelt werden. Hier soll sich zeigen, ob die eigene Lösung genauso schlank ist wie die JPA-Datenermittlung.

	eigene Entwicklung	**Entwicklung mittels JPA**
Verbindungsaufbau	426 ms	2550 ms
Summen-Abfrage	3 ms	45 ms
Listen-Abfrage	2665 ms	1255 ms

Tabelle 5.5: Performance-Vergleich der Dauer bei Datenbankabstraktion

Tabelle 5.5 zeigt, dass der Verbindungsaufbau bei einer reinen JDBC-Verbindung erheblich schneller ist. Auch ein direktes SQL-Statement kann in seiner Performance nicht von JPA geschlagen werden. Lediglich bei der Abbildung in POJOs zeigt sich, dass JPA der eigens entwickelten Lösung voraus ist.

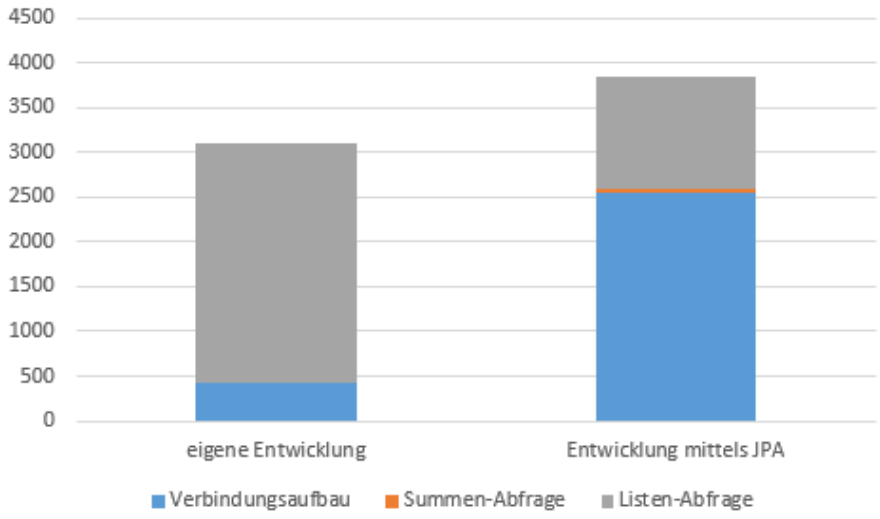


Abbildung 5.12: Performancevergleich der Dauer bei Datenbankabstraktion

Auch beim Ressourcenverbrauch zieht sich eine solche Linie durch. In Tabelle 5.6 sieht man, dass die Datenanbindung mittels JPA in jedem Punkt mehr Ressourcen verbraucht.

	eigene Entwicklung	Entwicklung mittels JPA
instanzierte Klassen	1529	3705
verbrauchter Arbeitsspeicher	14,37 MB	66,72 MB
durchschnittliche CPU Last	36,4 Prozent	54,4 Prozent

Tabelle 5.6: Performance-Vergleich des Ressourcenverbrauchs bei Datenbankabstraktion

Dass bei einer kleinen Applikation, bei der nur gerechnet wird, die CPU Last für ein kurzes Zeitfenster recht hoch ist, stellt kein Problem dar. Der massiv höhere Arbeitsspeicherverbrauch bei JPA könnte hingegen durchaus, bei großen Applikationen die auf älteren Rechnern laufen, zu einem Problem werden.

5.2.1.2 Fazit

Es zeigt sich, dass die direkte Datenbankanbindung ohne Verwendung einer zusätzlichen Technologie doch performanter ist. In zeitkritischen Bereichen ist es daher ratsam auf etwaige zusätzliche Technologien zu verzichten. Bei Endbenutzer-Applikationen hingegen, sollten die Abfragen nur einen Maximalwert nicht überschreiten, den der Benutzer als unangenehm empfinden würde. In diesem Fall liegen allerdings alle Zeiten unter dieser Schwelle. Ohne eine Stoppuhr würde ein Endbenutzer hier wahrscheinlich keinen Unterschied bemerken, daher sind beide Varianten für eine Endbenutzer-Applikation aus dem Sichtpunkt der Performance ausreichend.

Im Punkt Ressourcenverbrauch muss man den Vorwurf, dass moderne Technologien mehr Speicher und Performance benötigen, bei JPA gelten lassen. Allerdings sollte dieser Mehraufwand durch einen modernen Mittelklasse-Rechner ohne Probleme verkraftet werden können. Bei Kunden mit einer veralteten Infrastruktur sollte man erst experimentell ausprobieren, ob die Rechner die Mehrlast vertragen.

5.2.2 Tabelle Transaktionsliste

5.2.2.1 Performance

Hier wird die Transaktionstabelle aus den Testapplikationen in einen einfachen Frame eingebettet. Es wird zuerst analysiert, wie lang der Aufbau des Tabellen-Panels dauert und anschließend wie viel Zeit es in Anspruch nimmt dieses Panel in ein Fenster zu packen und anzuzeigen.

	eigene Entwicklung	Entwicklung mittels JPA und Java FX
Tabellenaufbau	3180 ms	4780 ms
Frame-Aufbau	89 ms	1086 ms

Tabelle 5.7: Performance-Vergleich der Dauer beim Tabellenaufbau

Wie Tabelle 5.7 und Abbildung 5.13 zeigen, bestätigt sich auch bei Java FX, dass eine moderne Technologie wesentlich mehr Zeit in Anspruch nimmt als eine klassische.

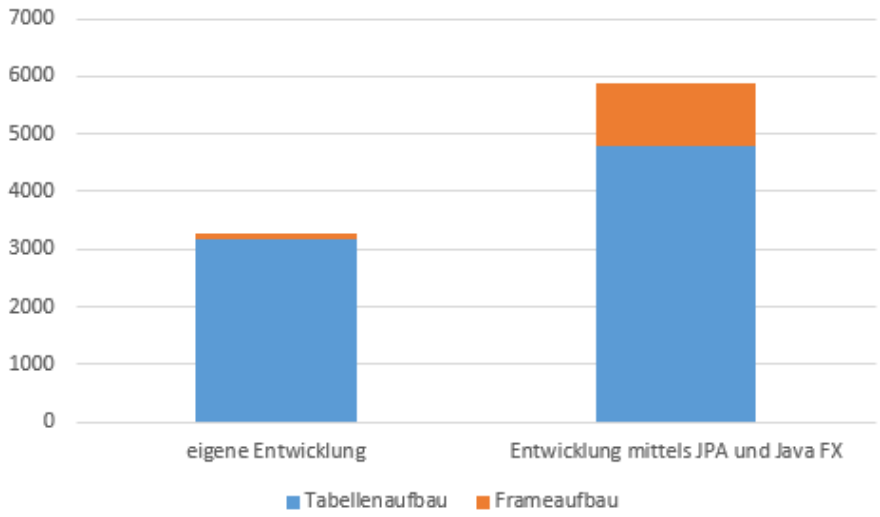


Abbildung 5.13: Performancevergleich der Dauer beim Tabellenaufbau

Laut Tabelle 5.8 fällt auch beim Ressourcenvergleich auf, dass die moderne Technologie um einiges mehr an Aufwand verursacht als die herkömmliche Herangehensweise.

	eigene Entwicklung	Entwicklung mittels JPA und Java FX
instanzierte Klassen	2674	5426
verbrauchter Arbeitsspeicher	18,50 MB	37,20 MB
durchschnittliche CPU Last	1,1 Prozent	0,2 Prozent

Tabelle 5.8: Performance-Vergleich des Ressourcenverbrauchs beim Tabellenaufbau

Interessant zu beobachten ist hier, dass der Speicherverbrauch bei der modernen Applikation unter dem Verbrauch des reinen JPA-Tests liegt. Das liegt daran, dass der GC[32] gleich nach der Suche nach allen Transaktionen eine Überlast an Speicherverbrauch wieder frei räumt. Abgesehen davon zeigt sich wieder, dass der Speicherverbrauch bei

[32] Garbage-Collector – räumt in Java automatisch nicht verwendeten Speicher wieder frei

den modernen Technologien wesentlich höher ist. Lediglich die CPU Last scheint bei beiden Anwendungen gleich verteilt zu sein.

5.2.2.2 Oberfläche

Ein Unterschied der sofort auffällt, ist dass die Java FX Tabelle von Haus aus sortiert werden kann – und zwar nach jeder beliebigen Spalte. Bei Java-Swing ist eine Sortierung – ohne eigene Entwicklung – nicht möglich. Abgesehen davon erscheint bei Java FX – wenn man das Fenster kleiner zieht – irgendwann ein vertikaler Scroll-Balken. Im Vergleich dazu gibt es bei Java-Swing nur einen horizontalen Scroll-Balken. Beim vertikalen Verkleinern des Fensters werden die Spalten immer dünner bis man den Inhalt nicht mehr erkennen kann. Bei beiden Tabellen lässt sich die Reihenfolge der Spalten austauschen und die Größe der einzelnen Spalten verschieben.

5.2.2.3 Fazit

Obwohl ermittelt wurde, dass die Datenermittlung mit JPA schneller läuft als die eigens entwickelte Lösung, dauert der Aufbau der Anzeige mittels Java FX um einiges länger als in der klassischen Applikation. Interessant ist die Tatsache, dass das einfache Hinzufügen des Panels in den Frame bei Java-Swing unter eine Zehntelsekunde dauert, während dieser einfache Schritt in Java FX bereits verhältnismäßig viel Zeit in Anspruch nimmt. Auch hier gilt – genauso wie bei JPA – das dieser Zeitunterschied für eine Endanwendung keine Rolle spielt. Allerdings muss auch hier darauf geachtet werden, dass das System mit genug Ressourcen ausgestattet ist.

Wenn es das System zulässt, sollte man auf jeden Fall moderne Technologien verwenden, da die Oberfläche nicht nur ansprechender aussieht, sondern standardmäßig viel mehr Features mit sich bringt.

5.2.3 Diagramm-Panel

5.2.3.1 Performance

Wie Tabelle 5.9 zeigt unterscheidet sich die Performance extrem stark von einander.

	eigene Entwicklung	**Entwicklung mittels JPA und Java FX**
Diagrammaufbau	420 ms	2944 ms
Frame-Aufbau	68 ms	649 ms

Tabelle 5.9: Performance-Vergleich der Dauer beim Diagrammaufbau

Da bei der eigens entwickelten Applikation das Diagramm komplett individuell entwickelt wurde und auf keine Technologien zurückgegriffen wurde, ist die Implementierung hier sehr schlank – was sich bei den Aufbauzeiten sofort widerspiegelt. Abbildung 5.14 zeigt, dass was die Zeiten angeht JPA und Java FX in diesem Fall überall das Nachsehen haben.

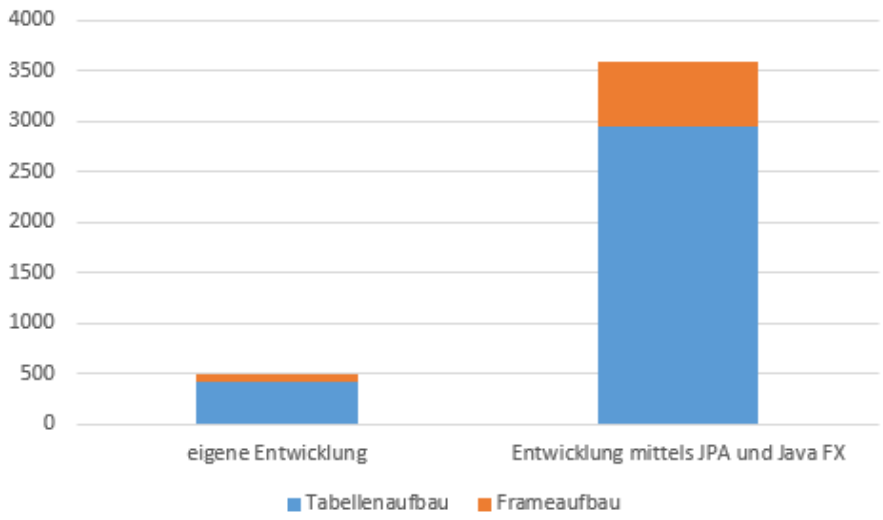


Abbildung 5.14: Performancevergleich der Dauer beim Diagrammaufbau

Auch beim Ressourcenvergleich zeigt laut Tabelle 5.10 die schlanke eigene Lösung ihre Wirkung.

	eigene Entwicklung	**Entwicklung mittels JPA und Java FX**
instanzierte Klassen	2291	5133
verbrauchter Arbeitsspeicher	7,33 MB	66,74 MB
durchschnittliche CPU Last	0,1 Prozent	2,6 Prozent

Tabelle 5.10: Performance-Vergleich des Ressourcenverbrauchs beim Diagrammaufbau

Der Speicherverbrauch – der bei der eingens entwicklten Variante fast nur ein Zehntel vom dem der modernen Applikation ist – ist schon beachtlich, vor allem wenn man bedenkt, dass beide Tabellen den selben Inhalt darstellen.

5.2.3.2 Oberfläche

Rein optisch ist die JPA Variante viel ansprechender. Die Balken haben abgerundete Kanten und einen Farbübergang während in der eigens entwickelten Anzeige ein Balken lediglich ein einfarbiges Rechteck ist. Auch der Hintergrund des Diagramms ist bei Java FX mit einem eingefärbten Rastermuster besser zu erkennen.

Außerdem passt sich das JPA-Diagramm horizontal der Größe des Fensters an – ein Feature welches in der Testapplikation nicht benötigt wurde und daher auch in dem eigens entwickelten Diagramm nicht enthalten ist.

5.2.3.3 Fazit

Hier zeigt sich, dass Lösungen die auf einen Bedarfsfall zugeschnitten sind immer performanter sind als die Allround-Lösungen aus den Technologien. Allerdings kann ein solch eigens entwickeltes Diagramm auch nur bei genau diesem einen Anwendungsfall benützt werden.

Grafisch ist das JPA-Diagramm dem eigenen stark überlegen. Auch wenn sich die Performance-Werte unterscheiden, liegen beide für heutige Maßstäbe im grünen Bereich.

Es kräftigt sich wieder die Aussage, dass bei performancekritischen Applikationen eigene Komponenten entwickeln werden sollten, allerdings bei allen anderen Applikationen, bei denen genügend Ressourcen zur Verfügung stehen, lieber auf moderne Technologien umgestiegen werden sollte.

5.2.4 Summen-Panel

5.2.4.1 Performance

Auch hier zieht sich in Tabelle 5.11 und Abbildung 5.15 die Linie fort, dass Java FX kontinuierlich mehr Aufbauzeit benötigt als Java Swing.

	eigene Entwicklung	**Entwicklung mittels JPA und Java FX**
Panel-Aufbau	549 ms	3293 ms
Frame-Aufbau	61 ms	298 ms

Tabelle 5.11: Performance-Vergleich der Dauer beim Summen-Panel-Aufbau

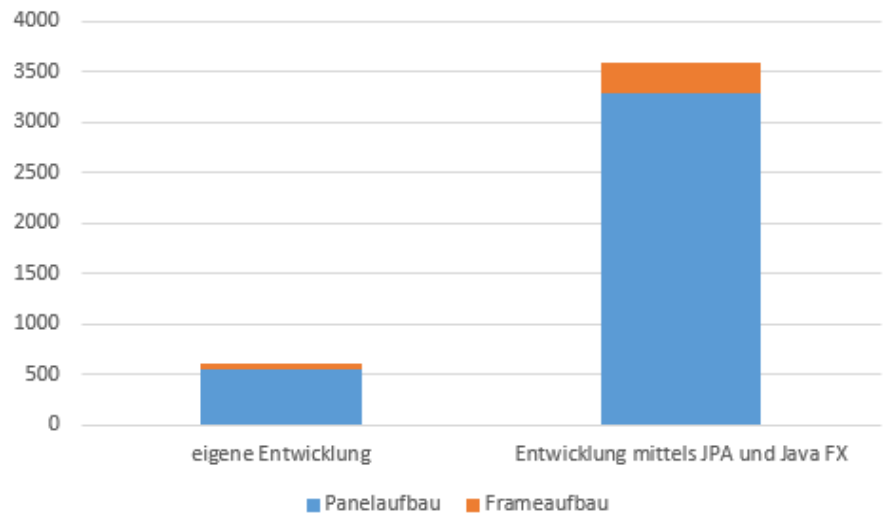


Abbildung 5.15: Performancevergleich der Dauer beim Summen-Panel-Aufbau

Tabelle 5.12 zeigt, dass die Ressourcenwerte sich wie gewohnt verhalten.

	eigene Entwicklung	**Entwicklung mittels JPA und Java FX**
instanzierte Klassen	2459	4919
verbrauchter Arbeitsspeicher	11 MB	66,7 MB
durchschnittliche CPU Last	0,3 Prozent	0,1 Prozent

Tabelle 5.12: Performance-Vergleich des Ressourcenverbrauchs beim Summen-Panel-Aufbau

5.2.4.2 Oberfläche

Zur Oberfläche lässt sich nicht viel sagen, da es sich in beiden Fällen um einfache statische Anzeigen handelt, die nicht auf die Fenstergröße reagieren.

5.2.4.3 Fazit

Dieses Panel sollte zeigen, ob es bei einfachen Anwendungsfällen grobe Unterschiede gibt. Und ja, man sieht, dass Java FX in jedem Fall mehr Speicher verbraucht. Wobei es interessant ist, dass der Arbeitsspeicherverbrauch bei einem kleinen Panel in Java FX genau so viel Speicher verbraucht, wie zum Beispiel ein Diagramm oder eine Tabelle. Wahrscheinlich werden bei Java FX immer eine gewisse Menge an Service-Klassen geladen, die in jedem Fall benötigt werden. Man wird sehen wie sich der Speicherverbrauch bei der gesamten Applikation verhält.

5.2.5 gesamte Applikation

5.2.5.1 Performance

Über die gesamte Applikation gesehen, relativiert sich das Ungleichgewicht in der Aufbauzeit ein wenig.

	eigene Entwicklung	**Entwicklung mittels JPA und Java FX**
Panel-Aufbau	3740 ms	4752 ms
Frame-Aufbau	90 ms	1381 ms

Tabelle 5.13: Performance-Vergleich der Dauer beim Applikation-Aufbau

Wie man in Tabelle 5.13 und Abbildung 5.16 sieht bleibt allerdings die klassische Variante in allen Punkten die schnellere Applikation.

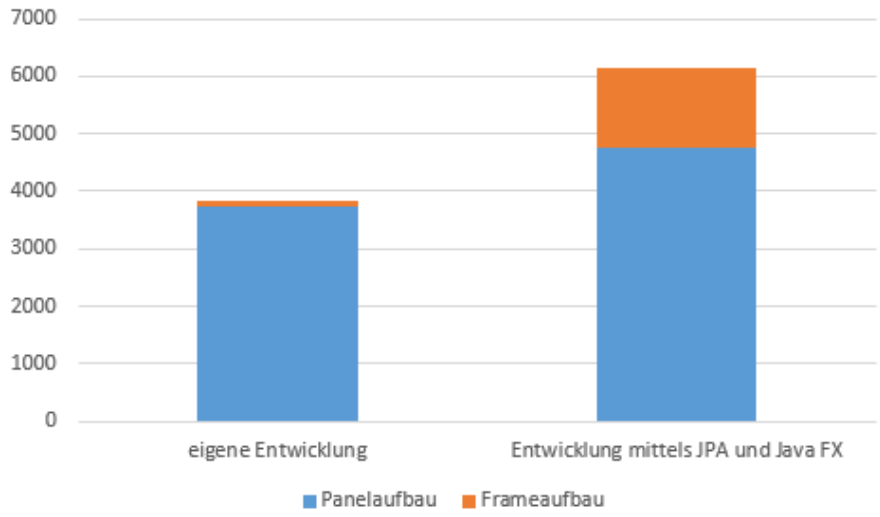


Abbildung 5.16: Performancevergleich der Dauer beim Applikation-Aufbau

Auch beim Ressourcenverbrauch zeigt Tabelle 5.14, dass über die gesamte Applikation gesehen der Abstand ein wenig kleiner wird, allerdings bleibt das bereits bekannte Schema bestehen, dass die klassiche Vorgehensweise weit weniger Ressourcen verbraucht.

	eigene Entwicklung	**Entwicklung mittels JPA und Java FX**
instanzierte Klassen	2699	5645
verbrauchter Arbeitsspeicher	19,58 MB	38,26 MB
durchschnittliche CPU Last	0,3 Prozent	0,3 Prozent

Tabelle 5.14: Performance-Vergleich des Ressourcenverbrauchs beim Applikation-Aufbau

Auch hier konnte der Effekt beobachtet werden, dass der Arbeitsspeicher in der modernen Applikation nach einigen Sekunden wieder leergeräumt und fast halbiert wird (siehe Abbildung 5.17). Es fällt auch auf, dass in beiden Fällen die gesamte Applikation circa gleich viel Arbeitsspeicher verbraucht wie ein einzelnes Panel.

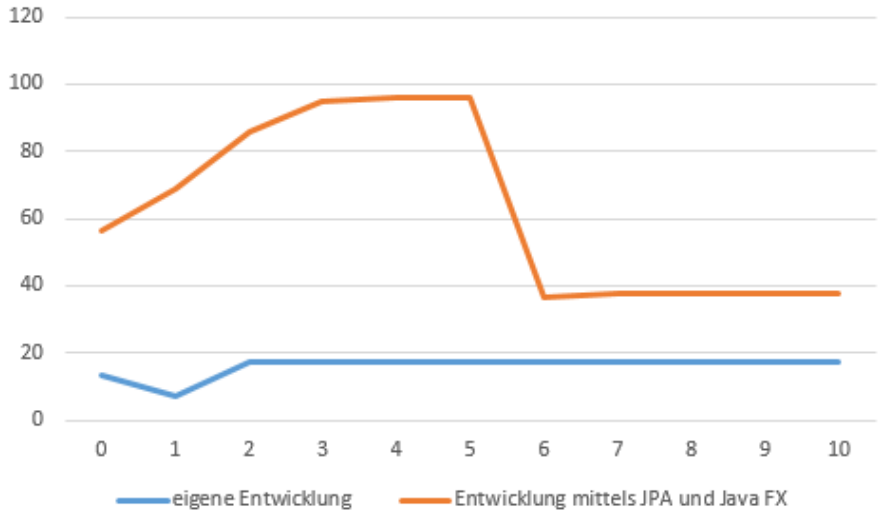


Abbildung 5.17: Performancevergleich des Speicherverbrauchs beim Applikation-Aufbau

5.2.5.2 Oberfläche

Die Spezialitäten der einzelnen Panels wurden bereits analysiert. Über die gesamte Applikation hinweg funktioniert in beiden Applikationen alles so wie es sollte. Die rein moderne Oberfläche von Java FX wirkt nur durch den Einsatz mehrerer Farben und Textgrößen ansprechender. Alle entwickelten Funktionen und Buttons reagieren über die gesamte Applikation gesehen richtig.

5.2.5.3 Release

DO-Applikation

Hier werden, da keine externe Technologie verwendet wird, lediglich die drei Archive

- DS.jar
- DO.jar
- mysql-connector-java-5.1.25-bin.jar

benötigt. Wobei in „DS.jar“ die geteilten Ressourcen (siehe Kapitel 4.1.2 und 4.1.3) enthalten sind. Der MySQL-Connector stellt den Treiber für die Datenbankverbindung zur Verfügung.

Die Größe der fertigen Applikation beträgt 986 KB.

DM-Applikation

Hier werden hingegen wie in Tabelle 5.15 dargestellt etliche Archive mehr benötigt.

Archiv	Beschreibung
DS.jar	enthält die geteilten Ressourcen
DM.jar	beinhalten die entwickelte Applikation
eclipseLink.jar	wird benötigt sobald EclipseLink verwendet wird
javafx-dialogs-0.0.3.jar	wird benötigt um Java FX Dialoge verwenden zu können
javax.persistence_2.1.0.v201304241213.jar	wird benötigt sobald auf JPA via EclipseLink zugegriffen wird
jfxrt.jar	wird von allen Java FX Applikationen benötigt
log4j-1.2.17.jar	wird benötigt da in dem Projekt Log4J zum Loggen verwendet wird
mysql-connector-java-5.1.25-bin.jar	stellt den Treiber für die verwendete MySQL Datenbank zur Verfügung

Tabelle 5.15: Benötigte Jar-Archive DM-Applikation

Die Größe der fertigen Applikation beträgt 22,4 MB.

5.2.5.4 Fazit

Was die Performance betrifft stimmen die Vorwürfe, dass die modernen Technologien mehr Speicherplatz verbrauchen und auch mehr Rechenzeit in Anspruch nehmen. Aufgrund der Erkenntnisse aus der Testapplikation kann man davon ausgehen, dass eine Applikation mit JPA und Java FX zwischen 30 und 50 Prozent mehr Ressourcen verbraucht als eine eigens entwickelte Lösung. Allerdings liegen die Rechenzeiten, als auch die verbrauchten Ressourcen noch immer in einem Bereich, der mit der heutigen Rechenleistung und deren Speichergrößen vertretbar ist. Wenn man eine alte Infrastruktur zur Verfügung hat, könnten diese beiden Technologien bei größeren Applikationen zu einem Performance-Problem führen. Es darf auch nicht unerwähnt bleiben, dass man aufgrund des höheren Performance-Verbrauchs auch mehr Features standardmäßig zur Verfügung hat.

Im Großen und Ganzen bleibt auch hier anzuraten auf die modernen Technologien umzusteigen, falls es die Infrastruktur zulässt. Bei zeitkritischen Applikationen sollte man

erst exemplarische Tests durchführen, ob JPA schnell genug ist. Java FX, da es sich um eine reine Endbenutzertechnologie handelt, ist nicht für den zeitkritischen Bereich gedacht und kann daher in jedem Fall empfohlen werden.

5.3 Erkenntnisse Erweiterbarkeitsvergleich

Hier wird anhand der beiden Beispiele aus Kapitel 4.4 analysiert, ob sich bei den unterschiedlichen Applikationen der Erweiterungsaufwand stark unterscheidet.

5.3.1 Datenbankänderung

5.3.1.1 Quantität der Entwicklung

Da die Testapplikation klein ist, konnte in beiden Fällen innerhalb von 30 Minuten die Applikation erfolgreich erweitert und getestet werden.

5.3.1.2 Qualität der Entwicklung

In JPA – da die Queries mit JPQL durchgeführt werden – musste hier nur eine einzige Programmzeile überarbeitet werden. Dabei handelt es sich um ein Architekturprinzip, welches auch bei größeren Applikationen nicht zu mehr Aufwand führt. Bei der DO-Entwicklung hingegen mussten in drei Klassen vier Zeilen ausgetauscht werden. Hier würde – da direkte SQL-Statements verwendet werden – der Aufwand bei größeren Applikationen ebenfalls steigen.

5.3.1.3 Fazit

Der Großteil des Aufwandes kann auch ohne moderne Technologien durch eine sinnvolle Architektur der Software vermieden werden. Allerdings muss hier der Entwickler in die Pflicht genommen werden, während bei JPA die Technologie selbst für eine sinnvolle Struktur sorgt. Bei einer eigens entwickelten Applikation, die in schlechtem Programmierstil geschrieben ist, wäre der Erweiterungsaufwand um einiges größer.

5.3.2 Sortierung der Konten in Java

5.3.2.1 Quantität der Entwicklung

Auch hier konnte kein Unterschied festgestellt werden. Bei einer übersichtlichen und modularen Struktur lassen sich auch herkömmliche Applikationen schnell erweitern.

5.3.2.2 Qualität der Entwicklung

Während in der herkömmlichen Applikation sechs Zeilen benötigt wurden, waren es bei der modernen Applikation lediglich zwei Programmierzeilen. Außerdem bleibt noch zu erwähnen, dass in der modernen Applikation komplett auf Java-Standard-Methoden zugegriffen werden konnte, sodass die Komplexität des Codes durch die Erweiterung nicht stieg. Im Gegensatz dazu musste bei der herkömmlichen Applikation ein eigener Work-Around entwickelt werden, da es in Java für „HashMaps“ keine Standard-Methoden zum Sortieren gibt.

5.3.2.3 Fazit

Es hat sich gezeigt, dass der Code mithilfe eines modularen Aufbaus auch in der herkömmlichen Entwicklung auf mögliche Erweiterungen vorbereitet werden kann. Damit ein Programmcode leicht erweiterbar bleibt, muss das Entwicklerteam in die Pflicht genommen werden übersichtlich und modular zu entwickeln. Moderne Technologien unterstützen dabei, allerdings bleibt die Hauptlast beim Entwickler. Somit kann man sagen, dass die Qualität eines Codes – und somit dessen Erweiterbarkeit – nicht von der Technologie, sondern vom Systemkonzept und Programmierstil abhängt.

6 Zusammenfassung der Ergebnisse und Ausblick

6.1 Zusammenfassung

6.1.1 Erkenntnisse bezüglich JPA

Die Einarbeitungszeit in JPA hat erheblich mehr Zeit in Anspruch genommen als erwartet. Das liegt nicht daran, dass die Technologie komplizierter ist als erwartet, sondern viel eher daran, dass die Möglichkeiten sehr vielseitig sind und es sich bei JPA um eine richtig ausgewachsene, vollständige Technologie handelt.

Die Implementierung mittels JPA erzeugt im Quellcode einen sich selbst dokumentierenden Code – da die Annotations beschreibend über allen Properties stehen und gleichzeitig deren Konfiguration vornehmen. Nach einer sinnvollen Einarbeitung kann man in JPA sehr performant mit der Entwicklung beginnen. Es kann mit minimalem Zeitaufwand eine große Datenbank abgebildet werden. Die Entwicklungszeit beschränkt sich in diesem Bereich nur auf einen Bruchteil der Entwicklungszeit mittels klassischen Vorgangsmustern. Das Befüllen und Auslesen der benötigten Datenobjekte ist bei JPA komplett automatisiert und fällt im Entwicklungsprozess daher komplett weg. Hier übernimmt der „EntityManger“ die meiste Arbeit.

Bezüglich der Performance muss man allerdings den Vorwurf gelten lassen, dass direkte SQL-Statements via JDBC-Treiber weniger Rechenzeit in Anspruch nehmen. Allerdings liegt das Delta zwischen der direkten Abfrage und den Abfragen mittels JPQL in einem Bereich, den man einem Endbenutzer mit menschlicher Reaktionszeit zumuten kann.

JPA hinkt auch beim Ressourcenverbrauch stark hinterher. Bei einem gängigen Computersystem sollte das kaum eine Rolle spielen, da der Ressourcenverbrauch zwar im Vergleich verliert, jedoch allgemein immer noch gut genug ist. Man sollte allerdings bei alten Rechensystemen aufpassen, ob man auf JPA umsteigen kann.

Da JPA – als Datenabstraktionstechnologie – auch in zeitkritischen Applikationen ohne UI[33] laufen kann, sollte man sich in diesem Bereich genau überlegen, ob es nicht sinnvoller ist auf eigene Lösungen zurückzugreifen, falls der Performance der Applikation die meiste Gewichtung zufällt.

JPA präsentiert sich als ausgereifte Technologie mit innovativen Ansätzen, die man in

[33] User Interface

jeder Linie nur weiterempfehlen kann.

6.1.2 Erkenntnisse bezüglich Java FX

Bei Java FX – im Unterschied zu JPA – hält sich die Einarbeitungszeit in Grenzen. Hier muss man nur sehr geringe Zeit in die allgemeine Einarbeitung (Konzeptverständnis) investieren und kann sich anschließend auf das benötigte Panel konzentrieren. Es konnte bereits relativ rasch eine erste Anwendung erstellt werden, deren Implementierung sehr leichthändig von statten ging.

Bei jeder Java FX Komponente gibt es einen „Builder“, der einen Entwickler ein Objekt sehr kompakt konfigurieren lässt und diesen dann in fertiger Form erzeugen kann. Durch die Verwendung dieser Builder können zwar sehr rasch komplexe Komponenten enstehen – die bei Java Swing mehrere Klassen benötigen würden – allerdings ist die Optik des Quellcodes für Java sehr gewöhnungsbedürftig. Das bedeutet nicht, dass der Code schlecht ist, ganz im Gegenteil: Nach einer Eingewöhnungsphase können diese Builder-Konstrukte viel fließender gelesen und verstanden werden als herkömmliche Komponentenkonfigurationen die sich möglicherweise über mehrere Klassen verteilen. Die Entwicklungszeit schrumpft auch hier, dank den Buildern, auf ein Minimum der Entwicklungszeit durch Java Swing. Das liegt daran, dass die Builder als komplette Framework-Komponenten angesehen werden können, die mittels Java Swing erst entwickelt werden müssten.

Ein weiterer sehr innovativer Schritt in Java FX ist die Verwendung von „ObservableLists“ – also Listen bei denen Java FX permanent auf eine Veränderung horcht. Somit muss bei einer Datenänderung, nicht wie in Java Swing, die gesamte Oberfläche manuell aktualisiert werden. Sobald man in einer dieser beobachteten Listen einen Wert hinzufügt, verändert oder entfernt, bemerkt Java FX diesen Zustand sofort automatisch und passt die Anzeige dahingehend an. Auch für die Änderung eines Wertes durch den Endbenutzer gibt es hierfür in Java FX – je nach Datentyp – verschiedenste beobachtete Properties, die ihre Werte in Java sofort anpassen.

Auch erwähnenswert ist, dass die Oberfläche in Java FX standardmäßig Features bereitstellt, die nicht entwickelt werden müssen. Zum Beispiel können Tabellen von Haus aus sortiert werden – ein Schritt, der in Java-Swing mit Aufwand verbunden ist.

Eine weitere wichtige Erkenntnis ist, dass Java FX noch nicht komplett ausgereift ist. Es sind bei der Technologieanalyse im Bereich Tabellenkonfiguration Fehler bei der Verwendung verschiedenster Datentypen aufgefallen. Bei der diesbezüglichen Analyse ist aufgefallen, dass die Dokumentation noch relativ klein ist und es kaum Code-Beispiele zu finden gibt. Allerdings konnte der Fehler schnell mit einem Work-Around umgangen werden, dieser stellt trotzdem keine saubere Lösung dar.

Auch bei Java FX gilt bei der Performance, dass die moderne Technologie nicht mit Java-Swing mithalten kann. Bei einem modernen Rechner sollte die erhöhte Rechenzeit zu keinem Problem führen – da sie sich auch bei Java FX in einem für den Endbenutzer erträglichen Bereich hält.

Im Gegensatz zu JPA kann man eine allgemeine Empfehlung aussprechen – da es sich bei Java FX um eine reine GUI-Technologie handelt, die aufgrund dieser Definition nicht in zeitkritischen Systemen zum Einsatz kommt. Es wird nur ausdrücklich davor gewarnt, dass es bei komplexen Anwendungen der Komponenten zu unerwarteten Fehlern kommen könnte – die leicht mittels zusätzlicher Handler und Listener manuell umgangen werden können. Allerdings gilt gerade deswegen, dass jede Implementierung unbedingt bis in die Tiefe getestet werden muss. Diese Kinderkrankheiten werden mit den nächsten Versionen wahrscheinlich ausgebessert werden.

6.1.3 gemeinsame Verwendung

Obwohl JPA und Java FX beide moderne Technologien sind die ihre Anfänge in einem ähnlichen Zeitraum hatten, sind die Herangehensweisen bei der Entwicklung doch sehr unterschiedlich.

Während in JPA fast zur Gänze mittels Annotations entwickelt wird – was zu einem sehr übersichtlichem Quellcode führt – setzt Java FX eher auf die eigens entwickelten Builder. Es ist interessant, dass in Java FX an keiner einzigen Stelle auf die äußerst sinnvollen Annotations zurückgegriffen wird. Trotzdem ist das Konzept der Builder sehr gut und die Anwendung schon fast intuitiv.

Einen gemeinsamen Nenner gibt es jedoch: Die Verwendung von POJOs und Properties. Ein JPA-POJO muss nur gering angepasst werden, um den beobachteten Proeprties von Java FX genüge zu tun. Diese JPA-POJOs können anschließend in eine beobachtete Liste – ObservableList – von Java FX geladen und an eine Komponente übergeben werden. Der „EntityManager“ aus JPA erkennt sofort Änderungen an den einzelnen POJOs und kann diese einfach in die Datenbank übertragen.

Auch wenn es eher wie ein Zufall scheint – da die Herangehensweisen bei den beiden Technologien doch sehr unterschiedlich sind – funktioniert die gemeinsame Entwicklung mit den Technologien sehr gut. Es kann ohne großen Aufwand eine Applikation erstellt werden, deren Quellcode sehr übersichtlich ist und deren Komplexitätsgrad sehr niedrig bleibt, da die meiste Logik in den Technologien steckt.

In Kapitel 5.1.5.5 wurde eine Formel entwickelt, die beschreibt, ab welcher Applikationsgröße sich die Einarbeitungszeit in die beiden Technologien lohnt. Das Ergebnis lautet: falls eine Applikation mittels herkömmlichem Entwicklungsweg über 170 Stunden Entwicklungszeit – je Entwickler – benötigt, sollte man aus rein wirtschaftlicher Sicht auf

die modernen Technologien umsteigen, da ab diesem Punkt durch die rasche Entwicklungszeit das Endprodukt trotz Einarbeitungszeit schneller fertig gestellt werden kann. Falls die Wirtschaftlichkeit nicht im Vordergrund steht, sollten moderne Technologien aus Sicht der Codequalität auch schon früher verwendet werden.

Bei der Performance allerdings hinken die modernen Technologien dem herkömmlichen Entwicklungsweg um circa 1/3 hinterher. Da es sich hier nicht um zeitkritische Endbenutzerapplikationen handelt – die der Reaktionszeit eines Endbenutzers unterliegen – spielt dieser Punkt beim Zusammenspiel der beiden Technologien JPA und Java FX keine größere Rolle.

Alles in allem kann gesagt werden, dass die Vielschichtigkeit und die Möglichkeiten der beiden analysierten Technologien überwältigend sind. Auch die Leichtigkeit der Entwicklung war in einem derartigen Ausmaß nicht zu erwarten. Um so überraschender war es, dass die Performance sich doch stärker unterscheidet als erwartet. Trotzdem lautet die allgemeine Empfehlung auf jeden Fall die Einarbeitungszeit zu investieren und bei der Entwicklung neuer Applikationen die modernen Technologien zu berücksichtigen. Die Qualität des Endproduktes – sowohl aus Sicht der Anwendung als auch des Quellcodes – steigt immens. Die Entwicklungszeit hingegen sinkt stark herab. Es gilt: höhere Qualität, niedrigere Quantität.

6.2 Ausblick

In dieser Arbeit wurden die Technologien JPA und Java FX lediglich im Hinblick auf die Entwicklung einer neuen Standalone-Rich-Client-Applikation analysiert. Jetzt da erfolgreich ermittelt werden konnte, dass sich ein Umstieg auf die modernen Technologien lohnt, kann der sichere und kleine Hafen der Standalone-Applikationen verlassen werden, um zu ermitteln, ob das Zusammenspiel von EJB und JPA in Enterprise-Applikationen auch dementsprechend gut funktioniert.

Abgesehen davon sind die meisten Applikationen heutzutage keine Neuentwicklungen, sondern basieren auf einem historischen Kern, der einen Umstieg auf neue Technologie schwer macht. Daher ist es wichtig zu ermitteln, ob ein Migration von alten Vorgehensweisen auf die modernen Technologien möglich ist, oder ob zumindest eine Koexistenz – in der neue Bereiche mit neuen Technologien entwickelt werden – möglich ist.

Literaturverzeichnis

[Gor] Gordon, Joni: *Working With Layouts in JavaFX*. http://docs.oracle.com/javafx/2/layout/builtin_layouts.htm.

[JLW14] James L. Weaver, Weiqi Gao, Ph.D. Stephen Chin Dean Iverson Wohan Vos Ph.D.: *Pro JaJava 2 - A definitive Guide to Rich Clients with Java Technology*. Apress, 2014.

[Mar12] Marco, Jakob: *JavaFX TableView Cell Renderer*, December 2012. http://edu.makery.ch/blog/2012/12/19/javafx-tableview-cell-renderer/.

[MW12] Müller, Bernd und Harald Wehr: *Java Persistance API 2*. Hanser, 2012.

[Ora13] Oracle.org: *Annotation Tutorial*. Internet, Oktober 2013. http://docs.oracle.com/javase/tutorial/java/annotations/.

[Org14] Organsiation, Oracle: *Java FX 2.2 API*, 2014. http://docs.oracle.com/javafx/2/api/.

[son13] sonarQuebe: *SonarQuebe Documentation*, Dezember 2013. http://docs.codehaus.org/display/SONAR/Documentation.

[War13a] Wartanian, Simon: *Modul Datenbanksysteme: BF-Projekt (JPA praktisch angewendet)*. Technischer Bericht, Hochschule Mittweida, 2013.

[War13b] Wartanian, Simon: *Praxisprojekt II - Framework KD2*. Technischer Bericht, Hochschule Mittweida, 2013.

Erklärung

Hiermit erkläre ich, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Mittweida, 1. März 2014

Anhang Verzeichnis

Beschreibung PDF-Merger 119

Test Diagramme 125

Zeitaufzeichnung 134

Anhang CD 135

PDF-Merger

Der PDF-Merger ist ein Programm, welches geschrieben wurde, um diese Diplomarbeit als ein gemeinsames Dokument abzuspeichern.

Der Großteil der Arbeit wurde in der Datei „main.pdf“ erstellt.

Allerdings wurde jeder Anhang als eigene Datei mittels Word 2013 geschrieben.

Ein Zusammenführen der Dateien mittels Word ist gescheitert (es wurde immer nur die erste Seite aller Dokumente kopiert).

Daher wurde ein Programm geschrieben, welches alle diese Dateien in richtiger Reihenfolge zusammenstückelt und ab den Anhängen eine fortlaufende Gesamt-Seitenzahl andruckt. Zusätzlich wird auf jeder Seite im Header der Titel des betroffenen Anhangs gespeichert.

Für den Merger wurde die Library iText in der Version 5.4.3 verwendet.

Hier der dokumentierte Quellcode des Mergers:

```
package merger;

import static merger.Constants.*;
import java.io.*;
import java.util.Scanner;
import com.itextpdf.text.*;
import com.itextpdf.text.pdf.*;

/**
 * Programm hängt PDFs in übergebener Reihenfolge aneinander
 * letzte Übergabe beschreibt das Output File
 * @author Simon Wartanian
 * @version       2013-08-15        WN       Erstelle Klasse
 */
final class PDFMerger {

        private final Font TEXT_NORMAL,
                           TEXT_GRAY,
                           HEADER;

        /**
         * @param args - INPUT-FILES dann OUTPUT-FILE
         * @throws IOException
         * @throws DocumentException
         */
        public static void main(String...args) throws IOException, DocumentException{
                Constants.OUTPUT.createNewFile();
                new PDFMerger();
        }

        /**
         * Konstruktor
         * @throws IOException
         * @throws DocumentException
         */
        private PDFMerger() throws IOException, DocumentException{
                System.out.println("starte Programm...");

                this.TEXT_NORMAL = FontFactory.getFont(FONT, 10, Font.NORMAL, BaseColor.BLACK);
                this.TEXT_GRAY = FontFactory.getFont(FONT, 10, Font.NORMAL, BaseColor.GRAY);
                this.HEADER = FontFactory.getFont(FONT, 20, Font.BOLD, BaseColor.BLACK);

                printPageNumberAndTitle(doIt());
                System.out.println("Zusammenhaengen erfolgreich abgeschlossen!");

                finish();
                System.out.println("temporaere Dateien wurden geloescht...");

                System.out.println("Druecke ENTER zum Beenden...");
                if(new Scanner(System.in).hasNextLine())
                        return;
        }
```

```
	private void finish() throws IOException{
		Constants.OUTPUT.delete();
		Constants.OUTPUT_DIRECTORY.delete();
	}

	private void writeAnhangDirectory(PdfCopy writer) throws DocumentException, IOException{
		System.out.println("schreibe Anhang-Verzeichnis...");

		Document doc = new Document();
		PdfWriter.getInstance(doc, new FileOutputStream(Constants.OUTPUT_DIRECTORY));

		doc.open();
		doc.newPage();
		Paragraph para = getHeader();
		para.add("Anhang Verzeichnis");
		doc.add(para);

		doc.add(getVerzeichnis());

		doc.newPage();
		doc.close();

		PdfReader pr = new PdfReader(new FileInputStream(Constants.OUTPUT_DIRECTORY));
		for(int i = 0; i < pr.getNumberOfPages(); i++)
			writer.addPage(writer.getImportedPage(pr, i + 1));
	}

	/**
	 *
	 * @return
	 */
	private Paragraph getHeader(){
		Paragraph para = new Paragraph();
		para.setLeading(50f);
		para.setSpacingAfter(10f);
		para.setAlignment(Paragraph.ALIGN_CENTER);
		para.setFont(HEADER);
		para.setKeepTogether(true);
		return para;
	}

	/**
	 *
	 * @return
	 */
	private Paragraph getHeaderFooterParagraph(){
		Paragraph para = new Paragraph();
		para.setAlignment(Paragraph.ALIGN_LEFT);
		para.setFont(TEXT_GRAY);
		para.setKeepTogether(true);
		return para;
	}

	public PdfPTable getVerzeichnis() throws DocumentException{
		PdfPTable table = new PdfPTable(2);
		table.setSpacingBefore(100f);
		table.setWidthPercentage(80f);
		table.setWidths(new float[]{.9f, .1f});

		boolean isFirst = true; //inhaltsverzeichnis
		for(AnhangHelper ah:ANHAENGE){
			// Inhaltsverzeichnis wird ausgelassen
			if(isFirst){
				isFirst = false;
				continue;
			}
			table.addCell(getFormattedCell(ah.getTitel()));
			table.addCell(getFormattedCell(
					(ah.getStartseite() - Constants.ROEMISCHE_SEITEN_ANZAHL + 1) + ""));
// + 1 wegen Inhaltsverzeichnis
		}
		return table;
	}

	private PdfPCell getFormattedCell(String text){
		PdfPCell cell = new PdfPCell();
		cell.setPaddingBottom(10f);
		cell.setBorderColor(BaseColor.WHITE);
		cell.addElement(new Phrase(new Chunk(text, TEXT_NORMAL)));
		return cell;
	}

	/**
	 * Logik
	 * @throws IOException
	 * @throws DocumentException
```

```
	 */
	private int doIt() throws IOException, DocumentException{
		System.out.println("starte Einlesen der Files...");

		Document doc = new Document();
		OutputStream os = new FileOutputStream(Constants.OUTPUT);
		PdfCopy writer = new PdfCopy(doc, os);

		doc.addAuthor(AUTHOR);
		doc.addCreationDate();
		doc.addProducer();
		doc.addCreator(AUTHOR);
		doc.addTitle(TITEL);
		doc.addSubject(GENRE);

		doc.open();

		System.out.println("starte Schreiben...");
		PdfReader pr = new PdfReader(new FileInputStream(INPUT));
		for(int i = 0; i < pr.getNumberOfPages(); i++)
			writer.addPage(writer.getImportedPage(pr, i + 1));

		int globalPageNumber = pr.getNumberOfPages()+1,		// wegen Anhang-Verzeichnis
			firstPageNumbers = pr.getNumberOfPages()+1;

		completeAnhangHelpers(globalPageNumber, firstPageNumbers); // wegen Anhang-Verzeichnis

		writeAnhangDirectory(writer);

		System.out.println("starte Zusammensetzen der Anhaenge...");
		for(AnhangHelper ah:ANHAENGE){
			if(ah.getFile().length() == 0)
				continue;
			pr = new PdfReader(new FileInputStream(ah.getFile()));
			for(int i = 0; i < pr.getNumberOfPages(); i++)
				writer.addPage(writer.getImportedPage(pr, i + 1));
			pr.close();
		}
		doc.close();
		writer.close();

		return firstPageNumbers;
	}

	/**
	 *
	 * @param globalPageNumber
	 * @param firstPageNumbers
	 * @throws FileNotFoundException
	 * @throws IOException
	 */
	private void completeAnhangHelpers(int globalPageNumber, int firstPageNumbers) throws
FileNotFoundException, IOException{
		boolean isFirst = true;
		for(AnhangHelper ah:ANHAENGE){
			if(isFirst){
				ah.setStartseite(globalPageNumber-1);
				ah.setSeitenanzahl(1);
				isFirst = false;
				continue;
			}
			PdfReader pr = new PdfReader(new FileInputStream(ah.getFile()));
			ah.setStartseite(globalPageNumber);
			ah.setSeitenanzahl(pr.getNumberOfPages());
			globalPageNumber += pr.getNumberOfPages();
			pr.close();
		}
	}

	/**
	 *
	 * @param firstPages
	 * @throws FileNotFoundException
	 * @throws IOException
	 * @throws DocumentException
	 */
	public void printPageNumberAndTitle(int firstPages)
			throws FileNotFoundException, IOException, DocumentException{
		System.out.println("starte Schreiben des Headers und Footers im Anhang...");
		// Seitenanz
		PdfReader pr = new PdfReader(new FileInputStream(Constants.OUTPUT));
		PdfStamper stamper = new PdfStamper(pr,
				new FileOutputStream(Constants.OUTPUT_FINAL.getAbsoluteFile()));
		PdfContentByte under = null;

		for (int page = firstPages; page <= pr.getNumberOfPages(); page++) {
```

```
                        String title = getRightTitle(page);
                        under = stamper.getUnderContent(page);
                        if(!title.equals("")){
                       // HEADER
                       ColumnText.showTextAligned(
                                under,
                                0,
                                new Phrase(title), 250, 790, 0);
                        }
                // FOOTER
                ColumnText.showTextAligned(
                        under,
                        0,
                        new Phrase("Gesamtseite " + (page - Constants.ROEMISCHE_SEITEN_ANZAHL)), 250, 40, 0);

                }
                stamper.close();
                pr.close();
        }

        /**
         *
         * @param page
         * @return
         */
        private String getRightTitle(int page){
                String retVal = "";
                for(int i = 0; i < ANHAENGE.size(); i++){
                        AnhangHelper ah = ANHAENGE.elementAt(i);
                        if(ah.getStartseite() < page)
                                if(ah.getTitel().equals(""))
                                        retVal = "";
                                else
                                        retVal = "Anhang " + i + ": " + ah.getTitel();
                }
                return retVal;
        }
}
```

Als Model wurde folgende Klasse verwendet:

```
package merger;

import java.io.File;

public class AnhangHelper {

        private final String titel;

        private final File file;

        private int seitenanzahl;

        private int startseite;

        public AnhangHelper(String titel, String pfad){
                this.titel = titel;
                this.file = new File(pfad);
        }

        public int getSeitenanzahl() {
                return seitenanzahl;
        }

        public void setSeitenanzahl(int seitenanzahl) {
                this.seitenanzahl = seitenanzahl;
        }

        public int getStartseite() {
                return startseite;
        }

        public void setStartseite(int startseite) {
                this.startseite = startseite;
        }

        public String getTitel() {
                return titel;
        }

        public File getFile() {
                return file;
        }
```

```
}
```

Etwaige Konstanten werden aus folgender Klasse ausgelesen:

```
package merger;

import java.io.File;
import java.util.*;

/**
 * Klasse liesst alle Konstanten ein
 * @author simon
 * @version        2013-10-24        WN        Erstellen der Klase
 */
public class Constants {
        public static String AUTHOR;

        public static String TITEL;

        public static String GENRE;

        public static String FONT;

        /**
         * Diplomarbeit Hauptfile
         */
        public static File INPUT;

        /**
         * Output fuer fertiges Gesamt-File
         */
        public static File OUTPUT;

        public static File OUTPUT_FINAL;

        public static File OUTPUT_DIRECTORY;

        public static int ROEMISCHE_SEITEN_ANZAHL;

        /**
         * Liste mit allen Anhaengen
         */
        public static final Vector<AnhangHelper> ANHAENGE = new Vector<AnhangHelper>();

        /**
         * liesst alle Konstanten ein
         */
        static{
                ResourceBundle res = ResourceBundle.getBundle("merger.Anhaenge");

                AUTHOR = res.getString("AUTHOR");
                TITEL = res.getString("TITEL");
                GENRE = res.getString("GENRE");
                FONT = res.getString("FONT");

                INPUT = new File(res.getString("PFAD_ARBEIT"));
                OUTPUT = new File(res.getString("PFAD_OUTPUT") + "_temp.pdf");
                OUTPUT_FINAL = new File(res.getString("PFAD_OUTPUT"));
                OUTPUT_DIRECTORY = new File(res.getString("PFAD_OUTPUT") + "_directory.pdf");

                String   titel = "TITEL_ANHANG",
                         pfad = "PFAD_ANHANG";

                // 1. Anhang ist immer das Anhangsverzeichniss
                ANHAENGE.add(new AnhangHelper("", ""));

                for(int i = 1; res.containsKey(titel + i); i++)
                        ANHAENGE.add(new AnhangHelper(res.getString(titel + i), res.getString(pfad + i)));

                ROEMISCHE_SEITEN_ANZAHL = Integer.parseInt(res.getString("ROEMISCHE_SEITEN_ANZAHL"));

                System.out.println("Einlesen der Properties erfolgreich abgeschlossen...");
        }
}
```

Gestartet wurde der Merger mit folgendem Property-File (Anhaenge.properties):

```
AUTHOR=Simon Wartanian
TITEL=KontoDaten 2
```

```
GENRE=Diplomarbeit
FONT=Times New Roman
ROEMISCHE_SEITEN_ANZAHL=15

PFAD_ARBEIT=C:/Users/simon/CloudStation/Weiz/Diplomarbeit/arbeit/Main.pdf
PFAD_OUTPUT=C:/Users/simon/CloudStation/Weiz/Diplomarbeit/arbeit/out.pdf

TITEL_ANHANG1=Praxisprojekt II - Framework KontoDaten2
PFAD_ANHANG1=C:/Users/simon/CloudStation/Weiz/PraxisprojektII/Semester2/Anhang1#PraxisprojektII_v2.pdf

TITEL_ANHANG2=Testbericht StarMoney
PFAD_ANHANG2=C:/Users/simon/CloudStation/Weiz/Diplomarbeit/anhaenge/programme/Anhang2#StarMoney9.pdf

TITEL_ANHANG3=Testbericht Steganos
PFAD_ANHANG3=C:/Users/simon/CloudStation/Weiz/Diplomarbeit/anhaenge/programme/Anhang3#Steganos.pdf

TITEL_ANHANG4=Testbericht WISO - Mein Geld
PFAD_ANHANG4=C:/Users/simon/CloudStation/Weiz/Diplomarbeit/anhaenge/programme/Anhang4#WISOMeinGeld.pdf

TITEL_ANHANG5=Beschreibung PDF-Merger
PFAD_ANHANG5=C:/Users/simon/CloudStation/Weiz/Diplomarbeit/anhaenge/Anhang5#PDFMerger.pdf

TITEL_ANHANG6=Test Diagramme
PFAD_ANHANG6=C:/Users/simon/CloudStation/Weiz/Diplomarbeit/anhaenge/Anhang6#TestDiagramme.pdf
```

Das Programm führt seine Aufgabe erfolgreich durch und terminiert anschließend:

```
Einlesen der Properties erfolgreich abgeschlossen...
starte Programm...
starte Einlesen der Files...
starte Schreiben...
schreibe Anhang-Verzeichnis...
starte Zusammensetzen der Anhaenge...
starte Schreiben des Headers und Footers im Anhang...
Zusammenhaengen erfolgreich abgeschlossen!
temporaere Dateien wurden geloescht...
Druecke ENTER zum Beenden...
```

DIAGRAMME

Im KontoDaten2 Programm soll es ein Diagramm geben, welches zu jedem Monat eine Gegenüberstellung von Einnahmen und Ausgaben darstellt. Dieses Diagramm soll vom KontoDaten2-Framework zur Verfügung gestellt werden.

Hierzu wird die gängige Library „JFreeChart“ getestet. Falls die Library den Anforderungen nicht entspricht, wird eine eigene Darstellung erstellt, da es sich hier um nicht allzu viel Aufwand handelt.

Es wird ein Diagramm erstellt, das über 18 Monate hinweg per Zufallsprinzip die Einnahmen und Ausgaben zwischen 500 € und 2000 € darstellt.

Da eine sinnvolle Darstellung in einem Fenster nicht möglich ist, soll es möglich sein im Diagramm horizontal zu scrollen.

Zum Übergeben der Daten wird die Klasse „BarChartObject“ verwendet:

```
/**
 * Klasse speichert zu jedem Monat die entsprechenen Einnahme- und Ausgabe-Summen
 * @author Simon Wartanian
 * @version         v0.1      17.02.2013          Erstellen der Klasse
 */
public class BarChartObject{

        /**
         * speichert das entsprechende Jahr mit Monat in folgender Form: yyyyMM
         */
        private final String yearAndMonth;

        /**
         * Variablem halten die Einnahme- und Ausgabe-Summen
         */
        private final int in, out;

        /**
         * Erstellt ein Objekt
         * @param pYearAndMonth - speichert Jahr und Monat in Form yyyyMM
         * @param pIn - Summe Einnahmen
         * @param pOut - Summe Ausgaben
         */
        public BarChartObject(String pYearAndMonth, int pIn, int pOut){
                this.yearAndMonth = pYearAndMonth;
                this.in = pIn;
                this.out = pOut;
        }

        /**
         * @return - entsprechendes Jahr und Monat (yyyyMM)
         */
        public String getYearAndMonth(){
                return yearAndMonth;
        }

        /**
         * @return - entsprechende Einnamen-Summe
         */
        public int getIn(){
                return in;
        }

        /**
         * @return - entsprechende Ausgaben-Summe
         */
        public int getOut(){
                return out;
        }
}
```

Programmcode 1: BarChartObjekt – Helper für BarChart

1.1 JFREECHART

JFreeChart ist eine gängige und sehr mächtige Diagramm-Library.

Es könnte nur sein, dass es Probleme bei der Darstellung gibt, da ich bisher nur Einsatzfälle kenne, bei denen das gesamte Diagramm in einem Bildbereich dargestellt wird.

1.1.1 VERSUCH UND OUTPUT

Getestete Version: JFreeChart 1.0.14 und JCommon 1.0.17

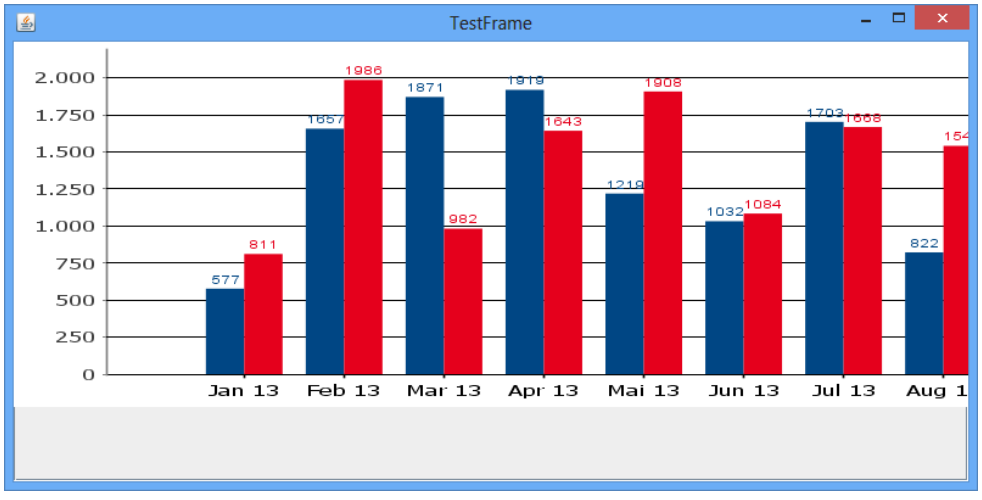
```
import java.awt.BasicStroke;
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.Paint;
import java.text.DecimalFormat;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Map.Entry;
import java.util.Vector;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.AxisLocation;
import org.jfree.chart.axis.NumberAxis;
import org.jfree.chart.labels.StandardCategoryItemLabelGenerator;
import org.jfree.chart.plot.CategoryPlot;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.chart.renderer.category.BarRenderer;
import org.jfree.chart.renderer.category.StandardBarPainter;
import org.jfree.data.category.CategoryDataset;
import org.jfree.data.category.DefaultCategoryDataset;
import org.jfree.ui.RectangleInsets;


/**
 * Erstellt ein JFreeChart-Diagramm mit Testdaten
 * @author Simon Wartanian
 * @version         v0.1      17.02.2013          Erstellen der Klasse
 */
public class TestJFreeChart{

          /**
           * Farben für die Balken
           */
          private final Color COLOR_PLUS = new Color(0, 70, 132),
                                                    COLOR_MINUS = new Color(229, 0, 28);

          /**
           * Schriftart und Größe die verwendet werden soll
           */
          private final Font  chartFont = new Font("Arial", Font.PLAIN, 9);

          /**
           * beschreibt Art der übergebenen Summe
           */
          private enum TYPE{IN, OUT};

          /**
           * speichert zu jedem Monat und Jahr die IN und OUT Werte
           */
          private LinkedHashMap<String, HashMap<TYPE, Integer>> barChartValues;

          /**
           * Beschreibt Monate des Jahres
           */
          private enum MONTH{
                    Jan, Feb, Mar, Apr, Mai, Jun, Jul, Aug, Sept, Okt, Nov, Dez;

                    /**
                     * Wandelt numärische Monate in Wortlaut des Monats um
                     * @param pMonth - Monat als Zahl
                     * @return - Monat im Wortlaut
                     */
                    public static MONTH getMonth(int pMonth){
                              return pMonth == 1 ? Jan : pMonth == 2 ? Feb :
                                        pMonth == 3 ? Mar : pMonth == 4 ? Apr : pMonth == 5 ? Mai :
                                                  pMonth == 6 ? Jun : pMonth == 7 ? Jul : pMonth == 8 ? Aug :
                                                            pMonth == 9 ? Sept : pMonth == 10 ? Okt :
                                                                      pMonth == 11 ? Nov : Dez;
                    }
          };

          /**
           * Erstellt Test-Frame
           * @param args - null
           */
          public static void main(String[] args){
                    // Frame mit der Auflösung 800 x 400 wird erstellt
                    JFrame frame = new JFrame("TestFrame");
```

```
		frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		frame.setLocationRelativeTo(null);
		frame.setLayout(new BorderLayout());
		frame.setSize(800, 400);

		// Vector mit BarChart-Objekten wird erzeugt
		Vector<BarChartObject> v = new Vector<BarChartObject>(){{
			int year = 2013;
			int month = 1;

			for(int i = 0; i < 18; i++){
				if(i == 11){
					year++;
					month = 1;
				}
				String yearAndMonth = year + "" + (month < 10 ? ("0" + month++) : month++);
				int in = 500 + (int)(Math.random()*1500);
				int out = 500 + (int)(Math.random()*1500);
				add(new BarChartObject(yearAndMonth, in, out));
			}
		}};

		// JFreeChart-Diagramm wird erzeugt und an ScrollPane übergeben,
		// und anschließend im Frame angezeigt
		JPanel pl = new TestJFreeChart().getChartPanel(v);
		JScrollPane scroll = new JScrollPane();
		scroll.add(pl);
		frame.add(scroll, BorderLayout.CENTER);

		// Frame wird angezeigt
		frame.setVisible(true);
	}

	/**
	 * wandelt die LinkedHashMap "barChartValues" in das von JFreeChart benötigte Format um
	 * @return - Daten in einem CategoryDataset aufbereitet
	 */
	private CategoryDataset createDataset(){
		DefaultCategoryDataset dataset = new DefaultCategoryDataset();
		for(Entry<String, HashMap<TYPE, Integer>> entry:barChartValues.entrySet())
			for(Entry<TYPE, Integer>entry2:entry.getValue().entrySet())
				dataset.addValue(entry2.getValue(), entry2.getKey(), entry.getKey());
		return dataset;
	}

	/**
	 * @return - fertiges JFreeChart-Objekt
	 */
	public JFreeChart getChart(){
		// hole Daten im richtigen Container
		CategoryDataset dataset = createDataset();

		// erstelle Balkendiagramm
		final JFreeChart chart = ChartFactory.createBarChart(
				// Chart-Titel
				"",
				// x-Achse
				"",
				// y-Achse
				"",
				// Daten
				dataset,
				// Orientierung
				PlotOrientation.VERTICAL,
				// Legende
				false,
				//Tooltips
				false,
				// TODO: konnte nicht ermittelt werden, Doku ist kostenpflichtig
				false
		);

		// erstelel Diagramm, 0-Punkt links unten
		final CategoryPlot plot = chart.getCategoryPlot();
		plot.setRangeAxisLocation(AxisLocation.BOTTOM_OR_LEFT);

		// Labels in Y-Achse
		final NumberAxis rangeAxis = (NumberAxis) plot.getRangeAxis();

		// minimaler Abstand der Linien zum Oberen Border des Diagramms
		rangeAxis.setUpperMargin(0.1);

		//Hintergrund aendern
		plot.setBackgroundPaint(Color.WHITE);
		plot.setRangeGridlineStroke(new BasicStroke(1f));
		plot.setRangeGridlinePaint(Color.BLACK);

		//Balken formatieren
		plot.setRenderer(new MyBarRenderer());

		//Deaktivieren vom Border
		plot.setOutlineVisible(false);
		plot.setAxisOffset(RectangleInsets.ZERO_INSETS);

		//X-Achse:
		plot.getDomainAxis().setAxisLinePaint(Color.BLACK);
		plot.getDomainAxis().setTickMarkPaint(Color.BLACK);
		plot.getDomainAxis().setTickLabelPaint(Color.BLACK);

		return chart;
	}

	/**
	 * @param pBarChartObjects - Vector mit benötigten Daten
```

```
	 * @return - ein fertiges Diagramm-Panel auf Bais von JFreeChart
	 */
	public JPanel getChartPanel(Vector<BarChartObject> pBarChartObjects){
		this.barChartValues =
				new LinkedHashMap<String, HashMap<TYPE,Integer>>();
		for(final BarChartObject bco:pBarChartObjects)
			barChartValues.put(
				MONTH.getMonth(
					Integer.parseInt(
						bco.getYearAndMonth().substring(4))) +
						" " + bco.getYearAndMonth()
							.substring(2, 4),
						new HashMap<TestJFreeChart.TYPE, Integer>(){{
							put(TYPE.IN, new Integer(bco.getIn()));
							put(TYPE.OUT, new Integer(bco.getOut()));
			}});
		ChartPanel pl = new ChartPanel(getChart());
		pl.setSize(getDimension());
		return pl;
	}

	/**
	 * @return - gibt die berechnete Größe des benötigten Platzes zurück
	 * TODO: hier wird die Breite nicht optimal angezeigt
	 */
	private Dimension getDimension(){
		return new Dimension(90 + (90 * barChartValues.size()), 300);
	}


	/**
	 * Die Klasse sorgt für eine anschauliche Darstellung des Diagramms
	 * @author Simon Wartanian
	 * @versoin		v0.1	17.02.2013		Erstellen der Sub-Klasse
	 */
	private class MyBarRenderer extends BarRenderer{

		/**
		 * Konstruktor - setzte einfache einstellungen
		 */
		public MyBarRenderer(){
			setShadowVisible(false);
			((BarRenderer)this).setBarPainter(
					new StandardBarPainter());
			setItemMargin(0);
			//Text über Bars
			this.setBaseItemLabelGenerator(
					new StandardCategoryItemLabelGenerator(
							"{2}",
							new DecimalFormat("###")));
			this.setBaseItemLabelsVisible(true);
			this.setBaseItemLabelFont(chartFont);
		}

		/**
		 * beschreibt Darstellung der Balken
		 */
		public Paint getItemPaint(final int row, final int column){
			// Abhängig vom Balen, wird die dazupassende Farbe gesetzt
			if(row == 0)
				this.setBaseItemLabelPaint(COLOR_PLUS);
			else this.setBaseItemLabelPaint(COLOR_MINUS);
			return row == 0 ? COLOR_PLUS : COLOR_MINUS;
		}
	}
}
```

Teste JFreeChart

Output:



Test-Output mit JFreeChart

1.1.2 ERKENNTNIS

Mit JFreeChart ist es recht aufwendig ein einfaches Diagramm zu erstellen. Auch wenn es viele Konfigurationsmöglichkeiten gibt, sind diese doch relativ starr, verschachtelt und schwer zu erreichen.

Es gibt aber auch viele wichtige Eigenschaften, die scheinbar nicht konfigurierbar sind, wie zum Beispiel eine fixe Breite pro Balken, damit sich dann anschließend die Breite des Diagramms, anhand der Menge der enthaltenen Daten dynamisch verändert.

1.1.3 FAZIT

JFreeChart Diagramme sind darauf ausgelegt, dass dem Panel eine fixe Größe zugewiesen wird. Je nachdem wie viele Daten dem Diagramm zugewiesen werden, werden die Balken im Diagramm so angepasst, dass die Größe des Panels nicht überschritten wird.

Für KontoDaten2 ist diese Eigenschaft ein Ausschlusskriterium. Es ist wichtig, dass jeder Balken – unabhängig von der Menge der Daten – gleich groß ist. Ein Hin- und Herscrollen soll es ermöglichen über den Panel-Rand hinaus schauen zu können.

Es macht aber keinen Sinn, dass bei einer Anzeige über 3 Jahre jeder Balken nur noch ein paar Millimeter dick ist, und der dazugehörige Text nicht mehr angezeigt werden kann.

1.2 EIGENE DIAGRAMM-KLASSE

Da JFreeChart mit seiner Anzeige nicht überzeugen konnte, wird eine speziell auf die Ansprüche von KontoDaten2 maßgeschneiderte Diagramm-Klasse erstellt.

Die Klasse soll genauso wie JFreeChart mit Daten der Instanz „BarChartObjekt“ versorgt werden.

Für diese Variante ist keine weitere externe Library notwendig.

1.2.1 VERSUCH UND OUTPUT

Es wurde vorübergehend eine einfache Klasse erzeugt, die anschließend bei der Umsetzung in den weiteren Kapiteln verfeinert wird. Die Grundvoraussetzungen wie eine fixe Balkengröße und die Scroll-Fähigkeit sind aber schon vorhanden:

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.MouseInfo;
import java.awt.event.ActionEvent;
```

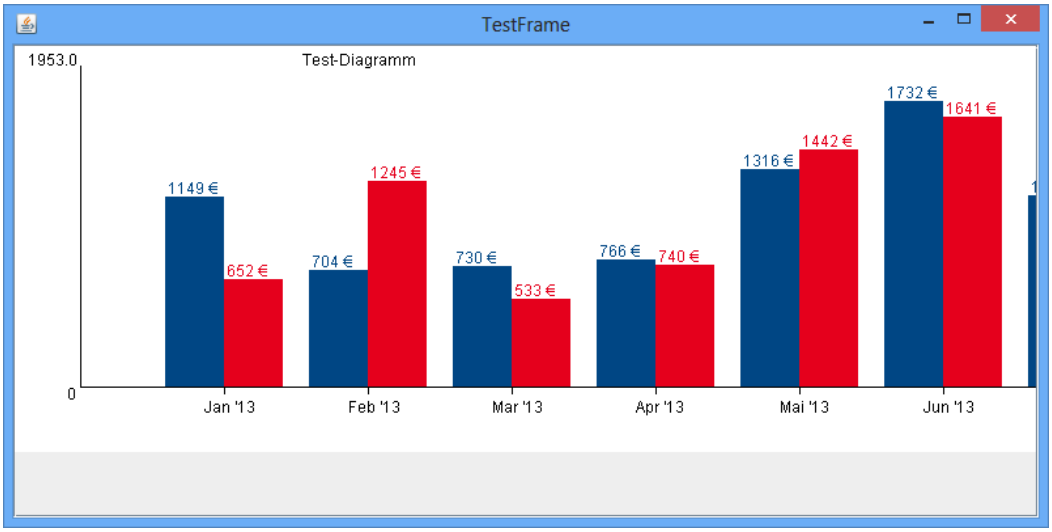
```
import java.awt.event.ActionListener;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.util.HashMap;
import java.util.Map.Entry;
import java.util.Vector;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.Timer;

/**
 * Erstelle ein einfaches Diagramm für KontoDaten2
 * @author Simon Wartanian
 * @version         v0.1      17.02.2013          Erstellen der Klasse
 */
public class TestEigenesBarChart extends JPanel{

          /**
           * Titel des Diagramms
           */
          private String title = "Test-Diagramm";

          /**
           * enthält alle Einnahme-Werte
           */
          private HashMap<String, Double> paintMap_plus;

          /**
           * Enthält alle Ausgabe-Werte
           */
          private HashMap<String, Double> paintMap_minus;

          /**
           * Enthält die maximale vorkommende Summe
           */
          private double maxSumme = 0;

          /**
           * beschreibt den Abstand zwischen den Monaten
           */
          private int intervall = 110;

          /**
           * beschreibt den Nullpunkt der Y-Achse in der Anzeige
           */
          private int yNull = 260;

          /**
           * beschreibt den maximalen noch angezigten Punkt der Y-Achse
           * zwischen yNull und yMax befinden sich alle Balkenhöhen
           */
          private int yMax = 15;

          /**
           * beschreibt den Nullpunkt der X-Achse, kann auch aus Anzeige rausrücken
           */
          private int xNull = 50;

          /**
           * beschreibt den maximalen Punkt der X-Achse,
           * kann sich auch außerhalb der Anzeige befinden
           */
          private int xMax = 0;

          /**
           * beschreibt die Breite eines Balkens
           */
          private final int bar_width = 45;

          /**
           * Beschreibt den Anfangswert, Nullwert
           */
          private final double xMin = 50;

          /**
           * beschreibt letzte Maus-Position um Delta berechnen zu können
           */
          private double oldMouseX = 0;

          /**
           * Beschreibt die Farben der Einnahmen- und Ausgaben-Balken
           */
          private final Color COLOR_PLUS = new Color(0, 70, 132),
                                                        COLOR_MINUS = new Color(229, 0, 28);

          /**
           * Erstellt ein Frame mit dem Diagramm
           * @param args - null
           */
          public static void main(String[] args){
                    // erstelle Frame mit der Größe 800 x 400
                    JFrame frame = new JFrame("TestFrame");
                    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    frame.setLocationRelativeTo(null);
                    frame.setLayout(new BorderLayout());
                    frame.setSize(800, 400);

                    Vector<BarChartObject> v = new Vector<BarChartObject>(){{
                              int year = 2013;
                              int month = 1;

                              for(int i = 0; i < 18; i++){
```

```
                                if(i == 11){
                                        year++;
                                        month = 1;
                                }
                                String yearAndMonth = year + "" + (month < 10 ? ("0" + month++) : month++);
                                int in = 500 + (int)(Math.random()*1500);
                                int out = 500 + (int)(Math.random()*1500);
                                add(new BarChartObject(yearAndMonth, in, out));
                        }
                }};

                // Panel wird erzeugt, eingebunden und angezeigt
                TestEigenesBarChart pl = new TestEigenesBarChart();
                pl.refreshMap(v);
                JScrollPane scroll = new JScrollPane(pl);
                frame.add(scroll, BorderLayout.CENTER);

                // Frame wird angezeigt
                frame.setVisible(true);
        }

        /**
         * Konstruktor
         */
        public TestEigenesBarChart(){
                // Diagram verschiebt sich mit Mauszeiger, kann allerdings nicht kleiner als xmin sein
                final Timer timer = new Timer(100, new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                                double currentX = MouseInfo.getPointerInfo().getLocation().getX();
                                if(oldMouseX != 0){
                                        double temp = xNull + (int)(currentX-oldMouseX);
                                        xNull = (int)(temp < xMin ? temp : xMin);
                                }
                                oldMouseX = currentX;
                                repaint();
                                System.out.println("Diagram.nullXPos = " + xNull);
                        }
                });

                this.addMouseListener(new MouseAdapter(){
                        private double oldX = 0;
                        public void mousePressed(MouseEvent e){
                                System.out.println("Start: verschiebe Diagram");
                                oldMouseX = 0;
                                timer.start();
                        }
                        public void mouseReleased(MouseEvent e){
                                System.out.println("Stop: verschiebe Diagram");
                                timer.stop();
                        }
                });
        }


        /**
         * wird bei jedem Repaint aufgerufen, hier wird das Diagramm gezeichnet
         */
        public void paint(Graphics g){
                // erstellen Weißes Rechteck, um altes Diagramm zu übermalen
                g.setColor(Color.WHITE);
                g.fillRect(0, 0, xMax + 100 > 700 ? xMax + 100 : 1200, yNull + 50);
                g.setColor(Color.BLACK);

                if(paintMap_plus != null){
                        //Y-Achse + Beschriftung
                g.drawString("0", xNull-10, yNull+10);
                g.drawLine(xNull, yNull, xNull, yMax);
                g.drawString(""+maxSumme, xNull-40, yMax);
                //Titel
                g.drawString(title, yNull-40, yMax);

                //X-Achse
                xMax = (paintMap_plus.size() + 1) * intervall;
                g.drawLine(xNull, yNull, xMax, yNull);

                //Intervalle
                Vector<String> tempV = new Vector<String>();
                for(Entry<String, Double> entry:paintMap_plus.entrySet())
                  tempV.add(entry.getKey());
                java.util.Collections.sort(tempV);
                System.out.println("Write Labels");
                for(int i = 0; i < tempV.size(); i++){
                  g.drawLine(xNull + intervall * (i+1), yNull-5, xNull + intervall * (i + 1), yNull + 5);
                  g.drawString(       mapYearAndMonth(tempV.elementAt(i)),
                                                  xNull + intervall * (i+1) - 15,
                                                  yNull + 20);
                  System.out.println(mapYearAndMonth(tempV.elementAt(i)));
                }

                //Lines
                int xMonth = xNull,
                  yMonth = yNull;
                for(int i = 0; i < tempV.size(); i++){
                  double buff_plus = yNull - ((yNull - yMax)/maxSumme) *
                                  paintMap_plus.get(tempV.elementAt(i)).doubleValue();
                  double buff_minus = yNull - ((yNull - yMax)/maxSumme) *
                                  paintMap_minus.get(tempV.elementAt(i)).doubleValue();
                  g.setColor(COLOR_MINUS);
                  g.fillRect((xMonth += intervall),
                                  (int)buff_minus, bar_width, yNull - (int)buff_minus);
                  g.drawString("" + paintMap_minus.get(tempV.elementAt(i)).intValue() +
                                  " €", xMonth + 2, (int)buff_minus - 2);
                  //
                  g.setColor(COLOR_PLUS);
```

```
                    g.fillRect(xMonth - bar_width,
                                    (int)buff_plus, bar_width, yNull - (int)buff_plus);
                    g.drawString("" + paintMap_plus.get(tempV.elementAt(i)).intValue() +
                                    " €", xMonth - bar_width + 2, (int)buff_plus - 2);
                    //
                    g.setColor(Color.BLACK);
                  }
                    }
          }


          /**
           * hier wird zu jedem String (yyyyMM) ein Monat im Wortlaut
           * zurückgegeben
           * @param pYearAndMonth - Jahr und Monat in Form yyyyMM
           * @return - Monat im Wortlaut
           */
          private String mapYearAndMonth(String pYearAndMonth){
                    String yy = pYearAndMonth.substring(2, 4);
                    int mm = Integer.parseInt(pYearAndMonth.substring(5));
                    String erg = "";
                    if(mm == 1)erg += "Jan";
                    else if(mm == 2)erg += "Feb";
                    else if(mm == 3)erg += "Mar";
                    else if(mm == 4)erg += "Apr";
                    else if(mm == 5)erg += "Mai";
                    else if(mm == 6)erg += "Jun";
                    else if(mm == 7)erg += "Jul";
                    else if(mm == 8)erg += "Aug";
                    else if(mm == 9)erg += "Sept";
                    else if(mm == 10)erg += "Okt";
                    else if(mm == 11)erg += "Nov";
                    else if(mm == 12)erg += "Dez";
                    return erg + " '" + yy;
          }

          /**
           * speichert übergebene Daten und zeigte diese anschließend an
           * @param transList - Vector mit benötigten Daten
           */
          public void refreshMap(Vector<BarChartObject> transList){

                    xNull = (int)xMin;
                    maxSumme = 0;
                    // plus und minus HashMaps werden neu erzeugt und befüllt
                    paintMap_plus = new HashMap<String, Double>();
                    paintMap_minus = new HashMap<String, Double>();
                    for(int i = transList.size()-1; i > -1; i--){
                              BarChartObject trans = transList.elementAt(i);
                              if(paintMap_plus.containsKey(trans.getYearAndMonth())){
                                        paintMap_plus.put(trans.getYearAndMonth(),
                                                                  new
Double(paintMap_plus.get(trans.getYearAndMonth()).doubleValue()));
                                        paintMap_minus.put(trans.getYearAndMonth(),
                                                                  new
Double(paintMap_minus.get(trans.getYearAndMonth()).doubleValue()));
                              }else{
                                        paintMap_plus.put(trans.getYearAndMonth(), new Double(trans.getIn()));
                                        paintMap_minus.put(trans.getYearAndMonth(), new Double(trans.getOut()));
                              }
                    }
                    // Maximum-Summe wird für die Anzeige ermittelt
        for(Entry<String, Double> entry:paintMap_minus.entrySet())
          if(maxSumme < entry.getValue().doubleValue())
                    System.out.println("maxSumme: " + (maxSumme = entry.getValue().doubleValue()));
                    for(Entry<String, Double> entry:paintMap_minus.entrySet())
                              System.out.println("Entry: " + entry.getKey() + " : " + entry.getValue());
        for(Entry<String, Double> entry:paintMap_plus.entrySet())
          if(maxSumme < entry.getValue().doubleValue())
                    System.out.println("maxSumme: " + (maxSumme = entry.getValue().doubleValue()));
                    for(Entry<String, Double> entry:paintMap_plus.entrySet())
                              System.out.println("Entry: " + entry.getKey() + " : " + entry.getValue());
          }
}
```

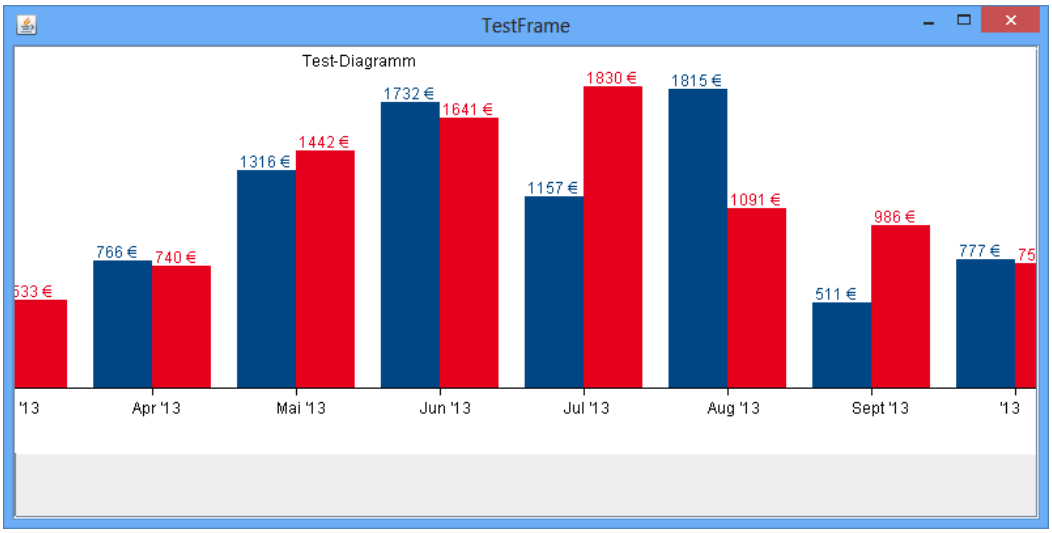
Eigene Diagramm-Klasse

Output:



Test-Output eigene Diagramm-Klasse

Output nach dem Verschieben (linke Mausetaste halten und Diagramm weiter „ziehen“):



Teste horizontales Scrollen mit eigener Diagramm-Klasse

1.2.2 Erkenntnis

Es lässt sich relativ schnell eine maßgeschneiderte Diagramm-Anzeige erstellen.

Es macht auch Sinn bei speziellen Anforderungen selbst eine Lösung auszuprogrammieren.

1.2.3 Fazit

Die eigene Lösung überzeugt und ist an Flexibilität uneingeschränkt. Es ist nicht mehr notwendig nach einer weiteren Lösungsmöglichkeit Ausschau zu halten.

KATEGORIE	THEMA	DATUM	VON	BIS	BESCHREIBUNG	Zusatz	gesamteArbeitszeit (T hh:mm):	05 20:00
Old Entwicklung	Table	21.02.2014	13:00	16:00	Entwickle FW-Tabelle			03:00
Old Entwicklung	Management	21.02.2014	16:00	16:30	Erstelle Logger-Klasse			00:30
Old Entwicklung	Management	22.02.2014	08:00	08:30	Loggingmeldungen einbauen			00:30
Architektur	GUI	22.02.2014	08:30	10:30	GUI-Definition in HTML			02:00
Old Entwicklung	DB	22.02.2014	10:30	13:00	Erstelle ID-Generator und DB-Connection und DB	DB_NAME: dipl_old, usr/pwd:diplomarbeit		02:30
Old Entwicklung	Event	26.02.2014	10:00	13:00	Erstelle Konzept und Klassen; Listener, Events und Bus			03:00
Analyse	Sonar	26.02.2014	19:30	22:00	Installiere Service; teste			02:30
Old Entwicklung	Event	26.02.2014	22:00	22:30	Arbeite an EventBus			00:30
Arbeit	JPA	01.03.2014	08:00	19:00	Lese JPA-Buch			11:00
Arbeit	JPA	02.03.2014	08:00	19:00	Lese JPA-Buch/Beginne mit Schreiben			11:00
Arbeit	JPA	03.03.2014	08:00	19:00	Lese JPA-Buch/Schreibe Erkenntnisse/Erstelle Programmbeispiele			11:00
Arbeit	JPA	04.03.2014	08:00	11:30	Assoziationen in JPA			03:30
Arbeit	JPA	04.03.2014	11:30	12:30	JPQL in JPA			01:00
Arbeit	JPA	04.03.2014	13:30	18:00	Lebenszyklus in JPA			04:30
Arbeit	JFX	05.03.2014	08:30	16:30	Beginne mit Einlesen und Testen von FX: Stage, Scene, Node			08:00
Arbeit	JFX	06.03.2014	08:30	13:30	Analyse Layouting; beginne mit Komponenten			05:00
Arbeit	JFX	06.03.2014	14:00	16:30	schließe Komponenten ab; beginne mit Tabelle			02:30
Arbeit	JFX	06.03.2014	18:30	22:00	arbeite an Tabelle -> analyse abgeschlossen, schreiben offen			03:30
Arbeit	JFX	07.03.2014	09:00	16:30	schreibe Tabellen; erarbeite und Schreibe Diagramme			07:30
Arbeit	JFX	08.03.2014	19:00	22:00	Erarbeite JavaFX Layouting			03:00
Arbeit	Systemkonzept	10.03.2014	08:00	16:00	Erstelle Konzept und schreibe dieses			08:00
Old Entwicklung	DB	11.03.2014	08:00	17:00	Erstelle Datenabstraktion			09:00
Old Entwicklung	Table	11.03.2014	17:30	23:00	Erstelle Tabellenlogik			05:30
Old Entwicklung	Diagramm	12.03.2014	08:00	15:00	Erstelle Diagrammlogik			07:00
Old Entwicklung	Summen	12.03.2014	15:00	18:00	Erstelle Summenpanel			03:00
Old Entwicklung	Applikation	12.03.2014	19:00	21:00	Führe Komponenten zusammen -> erstelle gesamte Applikation			02:00
Mordern Entwicklung	DB	13.03.2014	08:00	11:00	Erstelle Datenabstraktion			03:00
Mordern Entwicklung	Table	13.03.2014	12:00	16:00	Erstellen der Tabellenpanels			04:00
Mordern Entwicklung	Summen	14.03.2014	10:00	15:00	Erstellen des Summenpanels			05:00
Mordern Entwicklung	Table	15.03.2014	17:00	17:30	Zusammenschreiben			00:30
Mordern Entwicklung	Summen	15.03.2014	17:30	18:00	Zusammenschreiben			00:30
Mordern Entwicklung	Diagramm	15.03.2014	18:00	20:00	Entwickle Diagrammlösung			02:00
Modern Entwicklung	Applikation	16.03.2014	15:00	17:00	Erstelle DM Applikation			02:00
Old Entwicklung	Erweiterung	16.03.2014	17:00	17:30	Erstelle Erweiterungen			00:30
Modern Entwicklung	Erweiterung	16.03.2014	17:30	18:00	Erstelle Erweiterungen			00:30
Arbeit	Vergleich	16.03.2014	18:00	19:30	Erstelle Kapitelstruktur			01:30
Arbeit	Vergleich	18.03.2014	11:00	21:00	Erstelle Quellcodevergleich			10:00
Arbeit	Vergleich	19.03.2014	08:00	17:00	Stelle Vergleich fertig			09:00
Arbeit	Feinschliff	19.03.2014	17:00	19:00	Stelle Kapitel 1, 2, 3, 4 fertig			02:00
Arbeit	Zusammenfassung	20.03.2014	12:00	15:00	Schreibe Zusammenfassung, stelle Kapitel 5 fertig			03:00

Anhang CD

beinhaltet

- Quellcode der Applikationen
- Leere Datenbank
- Analyse-Quellcode
- Kompilierte Applikationen